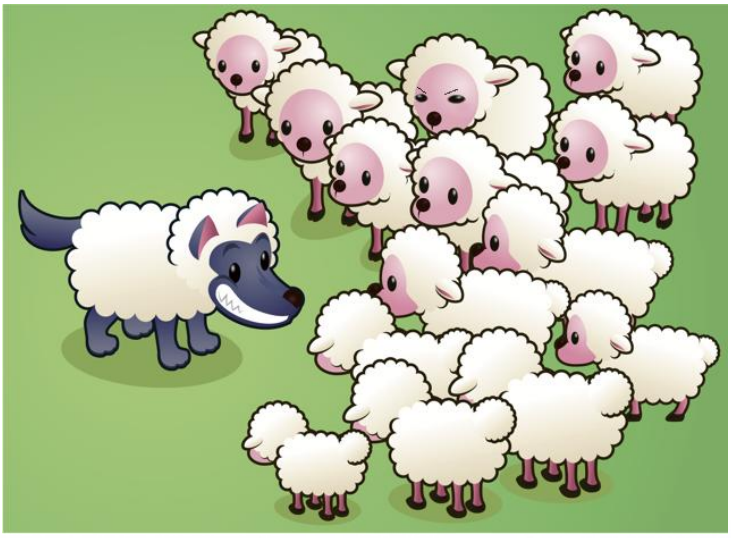
SANDIA REPORT

SAND2015-3711
Unlimited Release
Printed May 8, 2015

Counter Adversarial Data Analytics



Philip Kegelmeyer, Timothy M. Shead, Jonathan Crussell, Katie Rodhouse, Dave Robinson, Curtis Johnson, Dave Zage, Warren Davis, Jeremy Wendt, Justin “J.D.” Doak, Tiawna Cayton, Richard Colbaugh, Kristin Glass, Brian Jones, Jeff Shelburg

Prepared by
Sandia National Laboratories
Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy’s National Nuclear Security Administration under contract DE-AC04-94AL85000.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

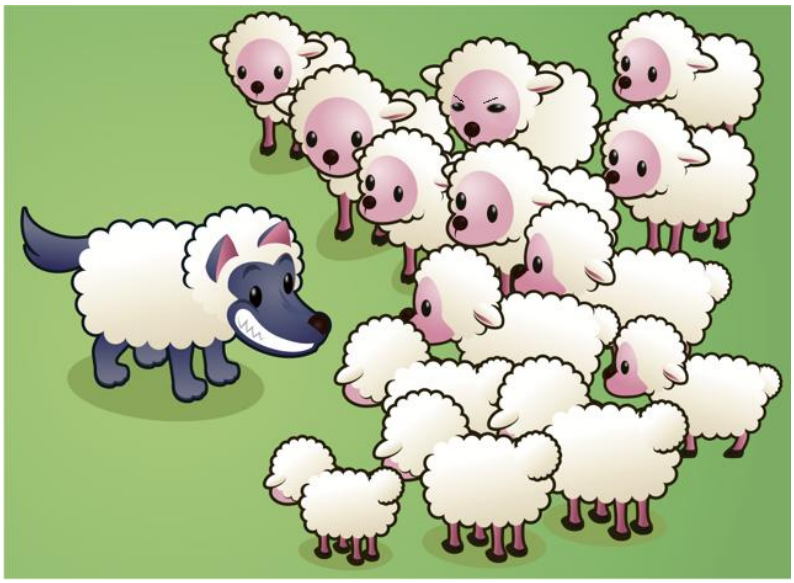
Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: http://www.osti.gov/bridge

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online

SAND2015-3711
Unlimited Release
Printed May 8, 2015

Counter Adversarial Data Analytics



Philip Kegelmeyer
Computer Sciences and Information Systems Center
Sandia National Laboratories
P.O. Box 969
Livermore, CA, 94551
wpk@sandia.gov

Timothy M. Shead, Jonathan Crussell, Katie Rodhouse, Dave Robinson,
Curtis Johnson, Dave Zage, Warren Davis, Jeremy Wendt,
Justin “J.D.” Doak, Tiawna Cayton, Richard Colbaugh,
Kristin Glass, Brian Jones, Jeff Shelburg

Abstract

This SAND report documents an LDRD effort, the "Counter Adversarial Data Analytics" (CADA) project. CADA was designed to develop and assess data analysis techniques intended to counter adversarial attacks against machine learning methods. Our motives for that specific focus were two-fold:

- Sandia makes critical use of data analysis to support high-consequence national security decisions. High-consequence issues often marked by the existence of adversaries who actively seek to evade or subvert that analysis.
- CADA was a brief, 1.5 year project. We therefore narrowed our scope to machine learning (which is admittedly only one of the many data analysis tools used by Sandia) as a data analytic that was specific enough to allow tractable analysis but powerful and broadly useful enough to have important Sandia application.

In the course of the CADA project we:

- demonstrated the counter-intuitive and alarming result that there exist label tampering attacks which are very effective while being nearly undetectable by the usual training data cross-validation tests,
- invented and thoroughly quantitatively investigated a novel method, "Ensembles of Outlier Measures" (EOM), for detecting and repairing tampered data,
- demonstrated the surprisingly generality of EOM as a data science principle by successfully applying it to an unsupervised problem very different from the one it was designed for, and
- developed a scheme for "quantified paranoia", that is, a statistically principled mechanism for deciding whether a dataset as a whole has been tampered with, regardless of whether we can detect any individual bits of tampering with confidence.

Acknowledgments

This work was funded under LDRD Project Number 171059 and Title “Counter-Adversarial Data Analytics”. Contributors no longer at Sandia include:

- Rich Colbaugh, Periander Ltd., rich.colbaugh@gmail.com
- Kristin Glass, Periander Ltd., kristin.glass63@gmail.com
- Brian S. Jones, Principal Research Engineer, FireEye, Inc., brian.jones@fireeye.com
- Jeff Shelburg, Google, jssdn2@mst.edu
- Yevgeniy (Eugene) Vorobeychik, Electrical Engineering and Computer Science, Vanderbilt University, Nashville, TN, yevgeniy.vorobeychik@vanderbilt.edu

Contents

Appendix

Figures

1 Introduction

Problem Statement

Sandia makes critical use of data analytics in defense of national security. Our adversaries therefore seek to sap, even suborn, those analytics. Through understanding our methods, they seek to produce data which is evolving, incomplete, deceptive, and otherwise custom-designed to defeat our analysis. Further, we cannot prevent them from doing so. We live in a changed world, in which we frequently *must* depend on data over which our adversaries have unprecedented influence.

This SAND report documents an LDRD effort, the "Counter Adversarial Data Analytics" (CADA) project, that was designed to develop and assess novel data analysis methods to counter that adversarial influence.

CADA was the inaugural activity funded by the Sandia Data Science Research Challenge process, and thereby had dual ambitions. First, it was to do data *science*, discovering generalizable and quantifiable counter-adversarial principles. Second, Sandia's national security mission requires methods that are *relevant*, applicable to analytics that matter, with realistic assumptions, useful uncertainty assessments, and practical implementations. As a result, CADA focused on machine learning specifically, as a data analytic that was specific enough to allow tractable analysis but powerful and broadly useful enough to have important Sandia application.

Basic Terms and Background

For the sake of a concrete running example of machine learning, consider an industrial inspection problem, one that examines samples on a production line and classifies them as "good" or "bad". Many modern industrial processes have analytics for extracting various features from a sample in order to predict subtle, future properties such as reliability or resistance to breaking. These analytics often employ *supervised machine learning*, which examines training data, "groundtruth" examples of good and bad samples, to generate rules for classifying future unknown samples.

For additional insight, it is often helpful to use the descriptive features to simply cluster samples that are similar, even without *a priori* labeling as good or bad. This is *unsupervised machine learning*.

Classic machine learning methods depend on two assumptions:

- First, that the groundtruth is indeed *true*, that it has not been poisoned ahead of time by deceptive samples, or tampered with afterwards by adversarial access.
- Second, that the eventual real data is *statistically similar* to the training data.

Both of those assumptions are violated[**?**] by an adversary who is trying to undermine our specific industrial inspection process, by corrupting part of the product groundtruth or by modifying

the "broken" samples specifically to confuse the current inspection system. We call this "label tampering".

Adversary Capabilities

Algorithm-aware label tampering is a usefully abstract and general problem formulation (see Section 4.1) but arguably also unrealistic or incomplete, in that it does not address any specifics of how an adversary might infer a defender's algorithms, or actually tamper with the data.

True enough. For CADA's research purposes (understanding the worst case effects of tampering, and designing counters), the precise mechanics of the attack perhaps don't strongly matter. As will be noted in the material to follow, most of our analyses trace out the effect of tampering across the full range of adversarial budget, and so guessing the actual budget from operational concerns is not required for our analysis.

Still, for some intuition into how these attacks might proceed, consider:

- An adversary wishing to infer our algorithms might start well simply by reading our published work, thus understanding what we think we well understand.
- More generally, they might simply brainstorm how they would optimally mount a defense themselves, then assume that the actual defenders were equally intelligent and had come to the same conclusion.
- One path to tampering with the data is an insider threat, or perhaps even network compromise. This is likely a low probability event, but there is historical precedent.
- Another path is realizing that analysis is more and more conducted on "found data", data taken in from the real world, and therefore it is possible to plant decoys (which we have taken to calling "data mines") to be collected and analyzed with naturally occuring data. We return to this specific scenario in Section 8.

Label Tampering as the Focus for Counter-Adversarial Data Analytics

One approach to addressing label tampering is simply to increase the robustness of the learning algorithm via methods such as regularization, minimization of worst-case loss, and compensation for "concept drift"[**?**]. These approaches are not particularly adversary aware, however.

Alternatively, we can harden machine learning models by improving the quality of their training data, specifically through detecting and addressing mislabeled "truth". Prior to CADA, Sandia had already demonstrated the ability to detect and correct *accidentally* mislabeled truth[**?**]. We also started with earlier anecdotal results suggesting that adversarial attempts to systematically mislabel data can paradoxically draw additional attention to the masked samples. The change in labels inescapably creates a signature, a change in the statistical nature of the "feature space", that

may mask individual samples but which nonetheless calls attention to the whole set of samples in that area.

In CADA we thoroughly investigated, generalized, and extended these findings. The general approach was to ask, repeatedly, from a variety of perspectives and on a variety of data:

- What is the possible effect of label tampering? How best to measure it?
- What would an adversary's goals be in tampering? Given those goals, what would an "optimal" tampering be? Is it computationally feasible to determine the optimum tampering? Are there useful heuristic approximations?
- Can we detect whether tampering has occurred at all? Can we detect which specific samples have been tampered with?

A Terse Summary of Results

Over the course of the CADA project we have:

- Exhaustively investigated the impact of adversarial label tampering on supervised machine learning, and in the process generated a durable body of re-usable Python code for future such investigations (Section A).
- We have demonstrated the counter-intuitive and alarming result that there exist label tampering attacks which are very effective (in the sense of decreasing test set accuracy near linearly with the number of points attacked) while being nearly undetectable by the usual training data cross-validation tests (Figure 6 and Section 4.3).
- The effectiveness of the various attacks we invented are, of course, worrisome. So we also invented, described (Section 5) and thoroughly quantitatively investigated (Section 6) "Ensembles of Outlier Measures", a method for building a meta-model for detecting tampered data. Further, we have shown how to use EOM to remediate tampered data by detecting, and correcting, the labels of suspect data points.
- In addition to working up a mechanism for individually detecting tampered points, we also created a method (Section 7) for "quantified paranoia", that is, a statistically principled mechanism for deciding whether a dataset as a whole has been tampered with, regardless of whether we can detect any individual bits of tampering with confidence. In tests we demonstrated (Table 12) that our quantified paranoia test does indeed find evidence of very limited tampering, even the tampering invisible to cross-validation, while not reacting to statistically similar untampered data.
- Finally, we have shown that the general principles behind Ensembles of Outlier Measures as a remediation technique are surprisingly and satisfyingly general. That is, in Section 8 we develop an entirely different sort of adversarial tampering problem, one in which an

adversary attempts to undermine an *unsupervised* machine learning method called DBSCAN by adding spurious elements to a data set. So the nature of the data, the nature of the analytic, and the nature of the attack are all very different from the supervised machine learning work investigated in most of CADA: yet an ensemble of outlier measures nonetheless turns out to be an effective means of detecting and removing the attack points, returning the clustering algorithm to its unattacked accuracy (Figures 31 and 32).

2 Prelude: A Review of Ensembles of Decision Trees (EDT)

Ensembles [**?**], and in particular ensembles of decision trees (EDT) [**?**], are a robust, reliably near-optimal method for practical, robust, accurate supervised machine learning[**?**]. They also effectively lower design stress in building a machine learning model, as they have the attractive property that they are unaffected by junk features and are invariant to monotonic transformation of the feature axes. Practically, this means there is no need to attempt to normalize features, to determine what their respective weight should be relative to each other: the decision tree ensemble can work this out on its own.

As a result, EDTs are very popular and widely used. We accordingly focused on EDTs as the primary supervised machine learning method investigated in CADA. We therefore begin with a review of the core nature of machine learning data, decision trees, and decision tree ensembles.

Training Data Format

DEFECT_ID	Defect? Truth	CGINTX a_1	CGINTY a_2	SNR a_3	... ...	PMIN a_K
d_1	Yes	12	1003	0.97	...	0.12
d_2	Yes	99	2	0.33	...	0.03
d_3	No	3	27	0.12	...	0.13
d_4	Yes	16	183	0.08	...	0.58
d_5	No	17	665	0.36	...	0.64
d_6	No	44	1212	0.29	...	0.42
d_7	No	42	24	0.33	...	0.88
d_8	Yes	78	42	0.44	...	0.52
⋮	⋮	⋮	⋮	⋮		⋮
d_N	No	12	3141	0.92	...	0.17

Table 1: Example Training Data, with Labels and Features

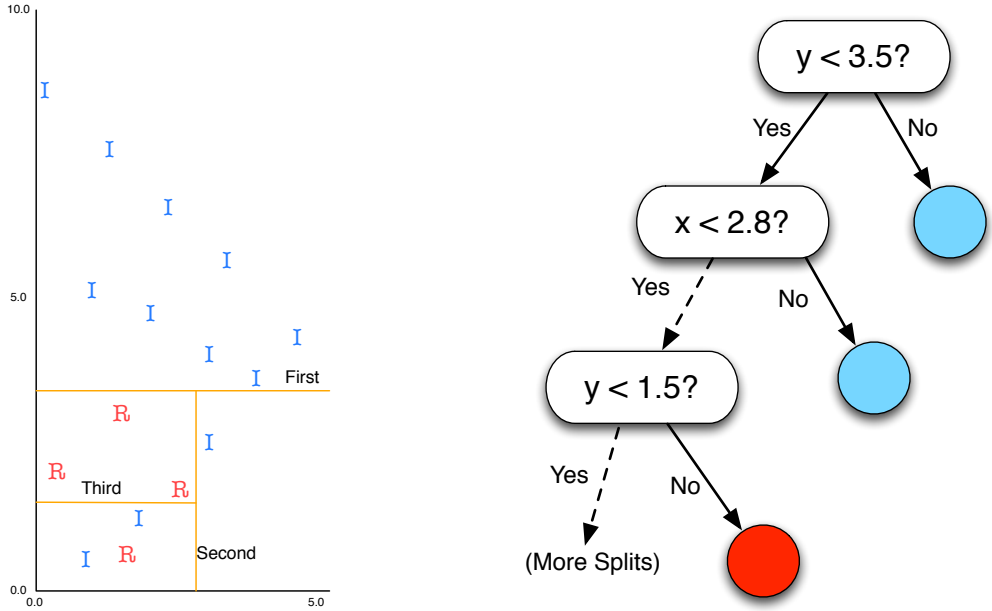
First, consider Table 1 (drawn from the NIF data described in Section 3), which illustrates the standard structure of training data suitable for supervised machine learning. Each row, each d_i, represents a single potential optical defect. The columns a_1 through a_K represent the K features that have been extracted to describe d_i. And since this is *training* data, data which has already been investigated and is understood, there is also a "Truth" column, indicating whether d_i is, or is not, a defect.

The core assumption behind supervised machine learning is that there is some function in the real world, $f(a_1,\ldots,a_k)$, that operates on the features to generate the labels, and the role of the machine learning algorithm is to generate an approximation of that function.

(A less frequently examined assumption is that the "Truth" labels are indeed *true*. Most of the rest of this report involves investigating the consequences of the violation of that assumption.)

Individual Trees

To understand how a single decision tree is built, take the geometric view of imagining each data point as existing in an N-dimensional space, where N is the number of attributes. (This is not strictly true if some of the attributes are not numerical, but the assumption suffices for this explanation.) Figure 1a illustrates a very simple case, with only two features (x and y), only 12 data



(a) Attribute space partitioned

(b) Decision tree representation

Figure 1: A 2D Attribute Space and its Decision Tree Partitioning

points, and two classes, Blue I and Red R. What the decision tree algorithm does is figure the best way to incrementally partition the feature space, giving each partition a unique label. Then to figure out the label of a *new* point, you figure out which partition it fell into, and look up its label. Or, equivalently, you march down the decision tree representation of the partition (as in Figure 1b) and look up the label associated with the leaf node of the tree.

To determine the optimal partitioning of the data, the tree algorithm defines an objective function (generally called a "purity" function) that measures how much more thoroughly the classes are separated in the children of any proposed split. Then it considers *all* possible splits, and picks the one which maximizes the objective function. Then the decision tree process recurses, and applies itself separately to whatever mixed class children remain, to figure out *their* best partition, and so on.

Ensembles of Trees

That's the process for building a single batch decision tree. There are many methods for building *ensembles* of classifiers. They can all be well enough illustrated by considering the "bagging" method, which is indeed the default ensemble method used in CADA.

The core idea behind ensembles via bagging is to generate many variants of a data set from an original baseline data set. Each variant is used to train a single machine learning model, and since each model is based on slightly varying data, the models will vary as well. Figure 2 illustrates the

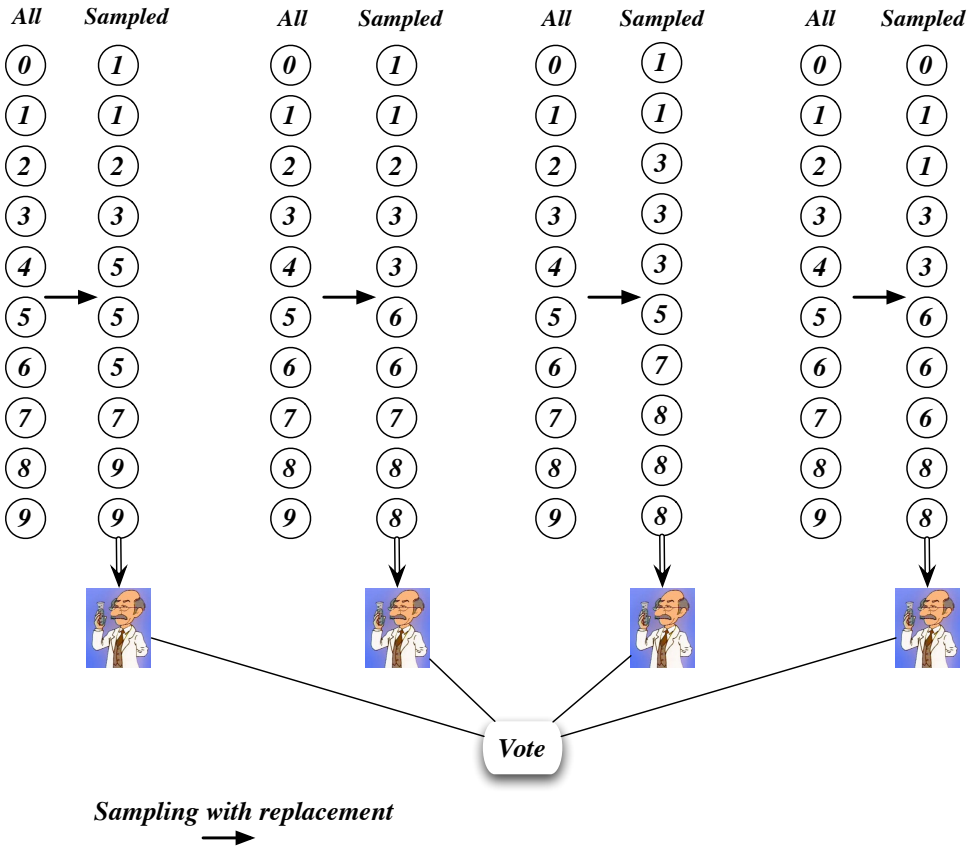


Figure 2: Four Parallel Examples of Bag and Tree Generation

core idea for a toy example which has only 10 training data points, labeled 0–9. Each of the four bagging examples depict picking randomly from those ten data points, *with replacement*, until ten points have been picked. Since the selection is done with replacement, any one data point might be picked more than once. And since a data point can be picked more than once, but the resultant data set is the same size, some points will be omitted. The result is a skewed, variant version of the original data. As an example, in Figure 2, the first bag has selected multiple copies of data points 5 and 9, but has omitted data points 4 and 6.

The point of ensemble methods is that simple majority voting of the resulting variant machine learning models is, nearly inevitably, more accurate than consulting a single model learned from

the unmodified data[**?**].

Avatar for EDT

The specific EDT implementation we used in CADA is "Avatar", a set of tools developed over earlier Sandia projects. The Avatar Tools embody all the algorithms and practices that have been gleaned from a decade's research into how to best make supervised machine learning work with huge, messy, noisy, unbalanced data.

Features that distinguish Avatar Tools from other "ensembles for decision trees" codes include:

- Native implementation of out of bag (OOB) validation, in addition to standard non-ensemble methods for cross-validation such as 10-fold and 5x2 validation.
- The use of OOB validation to automatically determine optimal ensemble size.
- Implementation of a wide range of skew-correction mechanisms, SMOTE[**?**] in particular.
- Implementation of a wide range of performance metrics.
- A post-processing analysis tool, "tree_stats", which provides a robust assessment of the relative importance of the features[**?**] used to build the EDT model. These assessments play a role in our investigation of remediation methods in Section 5.

3 Prelude: A Description of The Data

As mentioned, the intent of the CADA project was to arrive at *general* counter-adversarial principles, as pertains to machine learning. In particular, our aim was to avoid results that would be tied to the specific nature of the data under analysis.

Thus the larger context, the application details, of the data sets we used for research and test are not critical. Still, again for the sake of double checking that our results were indeed general, we acquired and operated on a variety of data sets from four application domains. Here we describe the four data sets we use in our supervised machine learning investigations, and give them the short name used to refer to them elsewhere in this report.

ideology: Earlier Sandia work[**?**] investigated whether documents could be classified as ideological through text analysis and machine learning. Each document was converted to a bag of words, and the resulting term-document vectors were reduced via PARAFAC2 tensor decomposition to convert each document to a vector of 200 continuous features, in a fashion similar to other term decomposition models such as Latent Semantic Analysis or Latent Dirichlet Allocation.

The "ideology" training data set contains 262 such vectors, labeled "Ideology" or "None" in half/half proportions. There is a matching test set of 262 vectors, again half/half, all distinct from the training set.

nif: The National Ignition Facility (NIF) at Lawrence Livermore National Laboratory is a high-energy laser facility comprised of 192 beamlines that house thousands of optics. The optics guide, amplify and tightly focus light onto a tiny target for fusion ignition research and high energy density physics experiments. The condition of those optics is critical to the economic, efficient and maximally energetic performance of the laser, but the number of optics and the required laser duty cycle means that human inspection of each optic after each laser shot is impossible. Reliable automated inspection is required, and so the NIF Optics Inspection (OI) project has accordingly developed sophisticated image analysis for the detection and description of potential defects on the optics[**?**]. These descriptions are in turn used as features fed to a machine learning model which identifies their nature.

We acquired such feature sets from the NIF OI project for analysis in CADA. They are usefully distinct in nature from the "ideology" data in that the twenty-eight features are a mix of categorical and continuous features, the continuous features are not necessarily commensurable in scale, and there are more than two classes of object.

Specifically, the "nif" training data consists of 300 such feature vectors, labeled "Not_a_defect", "Defect", or "Camera_defect" in third/third/third proportions. There is a matching test set of 300 vectors, again third/third/third, all distinct from the training set.

s500,b1000: The "ideology" and "nif" data sets are relatively modest in size, and though directly addressing scale issues was not a primary part of CADA's intent, we at least wanted to develop a sense of whether scale would prove a challenge in subsequent work.

We therefore also investigated data from a product inspection application. The "s500" training data consists of 500 vectors of twenty continuous features reflecting one way to inspect the underlying product. The "b1000" training data doubles that size, and consists of 1000 vectors of fifty continuous features reflecting a variant approach to characterizing the same products.

In both cases the labels were half/half "bad"/"good", and in both cases we created separate test data sets, of the same size and with the same half/half split in labels.

Finally, as described in detail in Section 8, we also investigated the attack and remediation of an *unsupervised* machine learning algorithm used to find plagiarized Android apps; it's data is described in Section 8.6.

4 Label Tampering, and its Consequences

4.1 The Adversarial Model

Adversary Goals

Over the course of this project, we modeled two distinct types of adversarial attack, which we term "confidence" and "evasion" attacks. They differ only in their goals; they both operate in the exact same fashion, which is to change some subset of the truth labels in a training data set *before* the supervised machine learning method can learn from it. With reference to Table 2, an attack is simply selecting a subset of $(d_1, \ldots, d_n)$ and flipping the label in the "Defect?" column to the opposite value.

DEFECT_ID	Defect? Truth	CGINTX a_1	CGINTY a_2	SNR a_3	... ...	PMIN a_K
d_1	Yes	12	1003	0.97	...	0.12
d_2	Yes	99	2	0.33	...	0.03
d_3	No	3	27	0.12	...	0.13
d_4	Yes	16	183	0.08	...	0.58
d_5	No	17	665	0.36	...	0.64
d_6	No	44	1212	0.29	...	0.42
d_7	No	42	24	0.33	...	0.88
d_8	Yes	78	42	0.44	...	0.52
⋮	⋮	⋮	⋮	⋮		⋮
d_N	No	12	3141	0.92	...	0.17

Table 2: Example Training Data, with Labels and Features

For a *confidence attack*, we assume the adversary seeks to reduce a defender's confidence in the efficacy of their machine learning models and algorithms, causing the defender to expend resources troubleshooting nonexistent problems or to abandon their models outright. We posited that such an adversary would seek to maximize the reduction in performance of a machine learning model on test data by tampering with the training data, while minimizing their own cost, i.e. minimizing the amount of tampering necessary to achieve a given performance reduction. In this case the adversary is unconcerned with the (mis)classification of individual observations in the test data; their sole purpose is to reduce the performance of the system as a whole.

Contrast this with an *evasion attack*[**?**], where we assume the adversary wishes to cause the misclassification of a single "payload" observation in the test data[**?**]. In this case the adversary is only tangentially concerned with the overall performance of the defender's model, so long as the payload observation is misclassified. Rather, they seek to achieve their goal with the least likelihood of detection, i.e. by minimizing the required training data tampering and avoiding any detectable reduction in system performance.

For both cases, we model an attack by determining:

1. A set of training data observations whose labels will be flipped (tampered with).
2. An explicit order in which the observations will have their labels flipped. (The order permits us to model an attacker's "budget"; she might be able to attack only n out of N samples, and so the ordering tells her which n to attack.)
3. An explicit "target" label that each tampered label will be flipped to. (In two-class problems this is obviously always the other class; in n-class problems the attack method must have some means of specifying which label to convert to.)

4.2 Confidence Attacks

The following outlines the confidence attack algorithms that we explored over the course of the project, roughly ordered from least- to most-sophisticated. Each algorithm was responsible for examining a training data set and producing an attack. For purposes of experimentation and insight, the attack algorithms were written to tamper with every observation in the test data. We will see in Section 4.3 that tampering with every observation is neither needed nor desirable in practice.

Random (random): The Random attack simply flips training observation labels in random order, providing a lowest-common-denominator baseline against which the other attacks can be compared.

Class Random (cr): For a training dataset containing L classes, the Class Random attack flips every label that matches class L_0 in random order, then every label that matches class L_1 in random order, and-so-on until every training observation label has been flipped.

Greedy Pessimal (gp): The Greedy Pessimal attack performs an iterative greedy search, flipping the training observation label that reduces test performance the most at each iteration until every training observation label has been flipped.

Brute Clustering (bc): For the Brute Clustering attack, the training observations are grouped into K clusters. Then, every label in a randomly-chosen cluster is flipped in random order. The process is repeated for the next randomly-chosen cluster, and this continues until every cluster has had all its observations flipped.

Subtle Clustering (sc): First, the training observations are grouped into K clusters. In addition to identifying the cluster that each observation belongs to, the clustering process provides the distance from each observation to the center of its cluster. Then, the Subtle Clustering attack flips every label in a randomly-chosen cluster, in decreasing order of label distance (i.e. the labels that are at the periphery of their cluster are flipped before labels that are closer to the center of the cluster). The process is repeated with another randomly-chosen cluster, until every cluster has had all its observations flipped.

Heterogeneous Clustering (hc): The training observations are grouped into K clusters, and the distance from each observation to the center of its cluster is noted. Heterogeneous Clustering then uses a one-way chi-squared test to attack the clusters in order from the most heterogeneous cluster to the most homogeneous. Within each cluster, labels are flipped in order from closest to furthest. Note that this was the opposite of the ordering of many of the other attacks; the intent being to attack the most homogeneous part of the cluster first, so as to sow confusion as quickly as possible.

Understated Clustering (uc): Understated Clustering groups training observations into K clusters, and the distance from each observation to the center of its cluster is noted. The clusters are then organized based on the percentage of labels within each cluster that match a target label L_t, in descending order. Within each cluster, labels are flipped in order from furthest to closest before moving to the next cluster. Here the motivation is to investigate an attack (still in the confidence attack context) that might prove useful in evasion attacks as well. The idea is that we first attack the clusters most populated by the label we are trying to hide, L_t, on the theory that those are the clusters where future L_t samples are likely to arise. Further, we attack from the outside in in the hopes of being relatively stealthy, as we are first creating anomalies in the region where anomalies are already relatively common.

Conditional Predictive Ordinates (cpo): A measure of the importance or influence of each observation relative to the classification of the population is provided by the Conditional Predictive Ordinate (CPO); see also Section 7.5. Attacks using the CPO strategy are based on one of two similar heuristics; both require the observations to be ordered based on the absolute magnitude of their individual CPO values. In a confidence attack, observation labels are flipped irrespective of which class label has been assigned. Alternatively, an evasion CPO attack involves the desire to intentionally bias the decision process associated with one or more specific labels, and so will attack only samples with certain labels, still ordered by absolute CPO.

4.3 Confidence Attack Experiments

Our confidence attack experiments were implemented using the Python [**?**] programming language. The implementations made extensive use of the NumPy and SciPy [**?**] modules for scientific computing. Individual experiments and results were combined in IPython [**?**] notebooks, and we used the IPython parallel toolkit to run experiments in parallel on multicore hosts. All of our experiments focused on the effects of confidence attacks on machine learning classifier accuracy; algorithm timings and efficiency were outside the scope of the project.

Each experiment used the following process (some steps were cached to avoid wasted computation):

1. Choose a dataset. Split the dataset into training and test observations, using random sampling without replacement.

2. Use statistics computed by the Avatar tree_stats post-processor to identify the two most important features in the training data. (The use of two features facilitated visualization of how the points were relabeled by the tampering process, and is not otherwise necessary. Spot checks using all the features did not show any fundamental differences in outcomes, though of course this should be revisited in any future work.)

3. Cluster the observations in the training data, using normalized versions of the two features identified in the previous step, the K-means clustering algorithm, and a heuristic for identifying the best value for K.

4. Identify a target (tampered) label for each observation in the training data. For two-class data, the tampered label for each observation obviously must be "the other class". For data containing n classes, we chose one of the other $n-1$ classes at random, with uniform probability.

5. Choose an attack algorithm from the list in the previous section. Compute attacks using the attack algorithm and the training data (and the clustering information, for those attack algorithms that required it). Most of the attack algorithms had a nondeterministic component, in which case twenty attacks were computed for each algorithm, each using different random seeds. A few attack algorithms were completely deterministic, in which case only one attack was computed.

6. Each individual attack was evaluated by generating an ensemble of decision trees using Avatar and the untampered training data, and collecting the out-of-bag accuracy on the training data and the voted and average accuracy of the ensemble on test data. This process was repeated after incrementally tampering with each label in the training data using the ordering and label targets computed by the attack algorithm, until every label in the training data had been tampered with. The resulting data can be best understood using the "degradation plot" of Figure 3, which plots the out-of-bag, voted, and average accuracy for an attack as the number of tampered labels increase; these degradation plots are discussed in detail in the next section.

7. From the available attacks, a "best" attack was selected using one of the metrics described later in this chapter.

8. The "best" attack was evaluated again, computing the same series of out-of-bag, voted, and average accuracy metrics as the tampering increased, but with a different random seed for the Avatar learner. This process was repeated twenty times, each time with a different Avatar seed. The data thus collected represented a range of possible outcomes for a single attack, in the presence of a defender using varying defenses.

Degradation Plots

A major tool for analyzing our confidence attack experiments was the "degradation plot" (Figure 3), which plots three measures of accuracy for ensembles of decision trees, as the number of tampered labels increase. Those three measures are as follows:

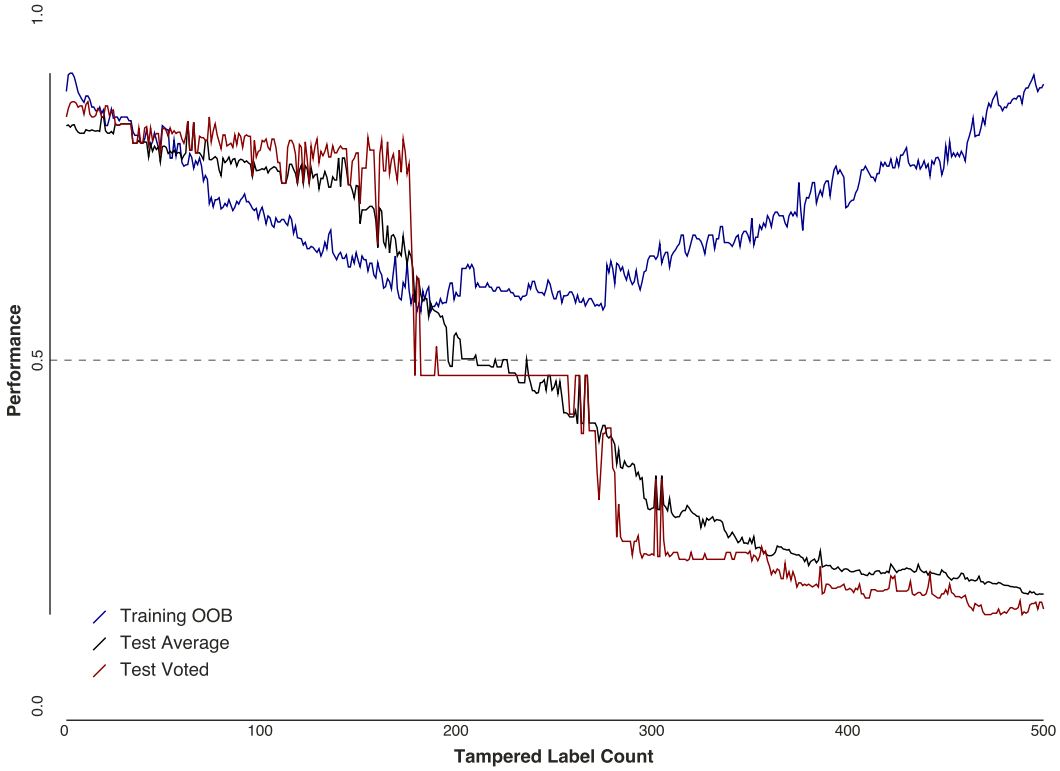


Figure 3: One random attack on the s500 dataset.

Out-of-bag: Out-of-bag accuracy (plotted in blue) is a type of cross-validation self-assessment available to ensembles of machine learning classifiers when each classifier is trained on a subset (the "bag") of the training data, and its performance is tested using the ("out-of-bag") remainder of the training data. As is always the case in machine learning, self-assessments are inherently optimistic, since they can only predict how well a model *might* generalize to new observations. Thus, under normal circumstances we expect the out-of-bag accuracy to be a little bit higher than other measures of accuracy, though we also expect the size of that gap to be relatively constant. It is important to note that we include out-of-bag accuracy in our results despite the challenges of self-assessment *because self assessment is the only kind of assessment a defender can make in the real world!*

Voted: Voted accuracy (plotted in red) is the accuracy of the ensemble as a whole, when the predictions of the individual classifiers are combined using voting and the group predictions are evaluated against test data. This represents the actual accuracy of the ensemble, when applied to new observations. Normally, we expect that the voted accuracy will be a little bit lower than the out-of-bag accuracy. Note that, because the voted accuracy can only be computed in experiments using test data that has ground-truth labels, it is unknowable for a defender in the real world.

Average: Average accuracy (plotted in black in degradation plots) is the average of the accuracy of each of the individual classifiers within an ensemble, when evaluated against the available test data. Thus, it serves as an approximation of the performance of a single classifier, if the defender wasn't using ensembles. Under normal circumstances, we expect the average accuracy to be lower than the voted accuracy of the entire ensemble (otherwise, there would be no point in creating ensembles). Like voted accuracy, the average accuracy cannot be known by a defender in the real world, since test data with ground-truth labels will not be available.

Average accuracy is a useful diagnostic for two reasons. First, it is a measure of how challenging the machine learning problem is: if voted and average accuracy are roughly equal, then the use of ensembles is not buying you much, the problem is so simple that a single tree suffices.

Secondly, if average accuracy is *higher* than voted accuracy, then this is a rare marker that something is seriously wrong. Usually it signifies that the errors in the individual trees are not sufficiently decorrelated, they are all making the same mistakes, which undermines the core principle of ensembles.

Applying these ideas to the degradation plot of a random attack in Figure 3, we see that in the absence of any tampering (at $x = 0$, at the far left end of the plot), out-of-bag (blue) accuracy is higher than voted (red) accuracy, which is higher than the average accuracy (black) for the dataset, as expected. Moving left-to-right we see that as the number of tampered training labels increase, all three measures decrease. However, note that for the first 170 tampered labels, the voted performance does not decrease as quickly as the average performance, indicating that the ensemble is affected less by tampering than a single classifier. However, after 170 labels have been altered, the ensemble performance quickly drops to 50 percent, where it is no better than a coin toss. Even worse, note that the ensemble performance stays below the average performance for most of the rest of the plot, i.e. that the defender in this case is paying the cost of creating and evaluating an ensemble of classifiers, yet achieving lower performance than a single classifier!

We now turn to the out-of-bag accuracy. During the first half of the attack in Figure 3 the out-of-bag accuracy drops quickly, even more quickly than the actual accuracy. This is ideal for the defender, since it can act as a strong signal than an attack is underway. However, the out-of-bag accuracy levels-out at around 60 percent until the attack reaches the halfway point, then returns to its original, untampered accuracy by the time the attack has tampered with every label in the training data. This highlights a danger in relying on self-assessment: as the amount of tampered data increases beyond 50 percent, the ensemble becomes increasingly adept at learning the new (tampered) truth; when its performance is evaluated against the same tampered truth, it gets good scores, and the self-assessment reflects this.

Of course, Figure 3 visualizes just one among many possible random attacks. To characterize the performance of an entire family of attacks (such as when we computed multiple attacks with varying random seeds), we created degradation plots showing envelopes of attack performance, as in Figure 4. The color scheme of the plot is unchanged, but now we plot the mean values of each measure across every attack, surrounded by a dark band representing the second and third

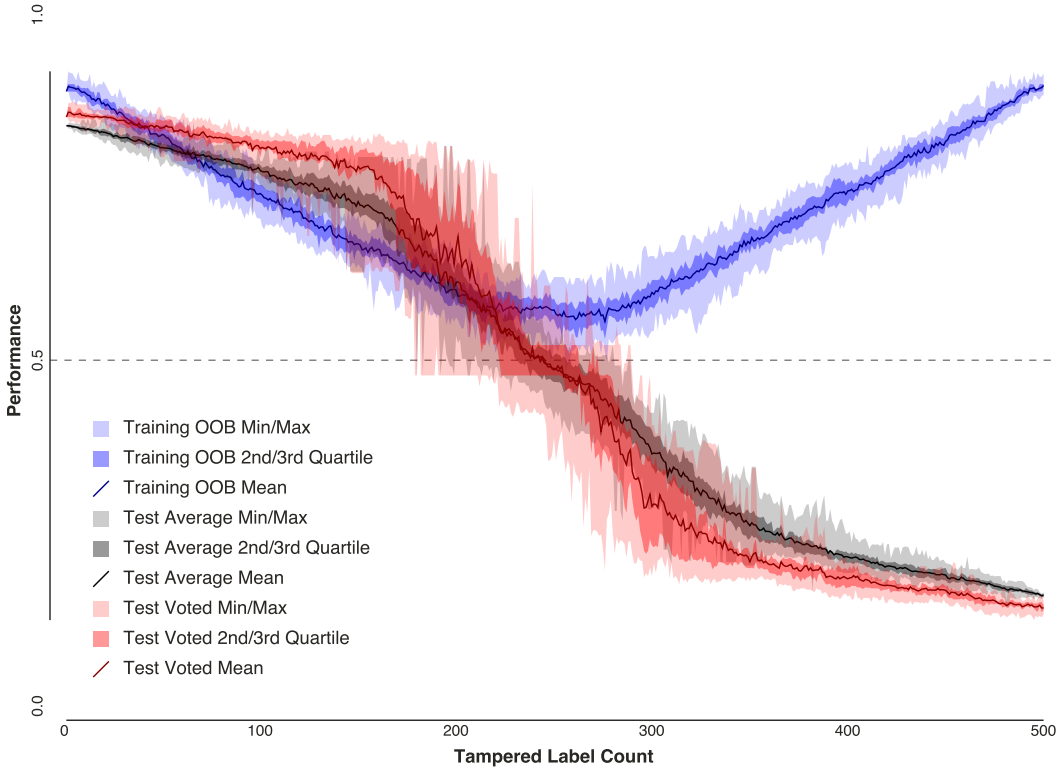


Figure 4: Twenty random attacks on the s500 dataset.

quartiles, surrounded by a lighter band representing the first and fourth quartiles, bounded by the minimum and maximum values. This gives a good sense for the distribution of values across the different attacks, and allows us to see that the trends described previously still apply to the various attacks. Wide areas in the bands indicate high variance in performance among attacks.

Using metrics described later, we typically chose one "best" attack from among the attacks for a given algorithm, then tested that attack against multiple defenses, i.e. multiple random seed parameters with the Avatar learner. This allowed us to evaluate how much the performance of a given attack might vary in the face of unknown defender parameters. Figure 5 is a degradation plot of a set of defenses against a single "best" random attack. Note that the variance of all three measures is much lower than that in Figure 4, indicating that for this combination of data set and attack algorithm, it was worthwhile for the attacker to take the time to choose a good attack (high variance in Figure 4), and that the attack chosen is likely to achieve relatively consistent performance even for a variety of defenses (low variance in Figure 5).

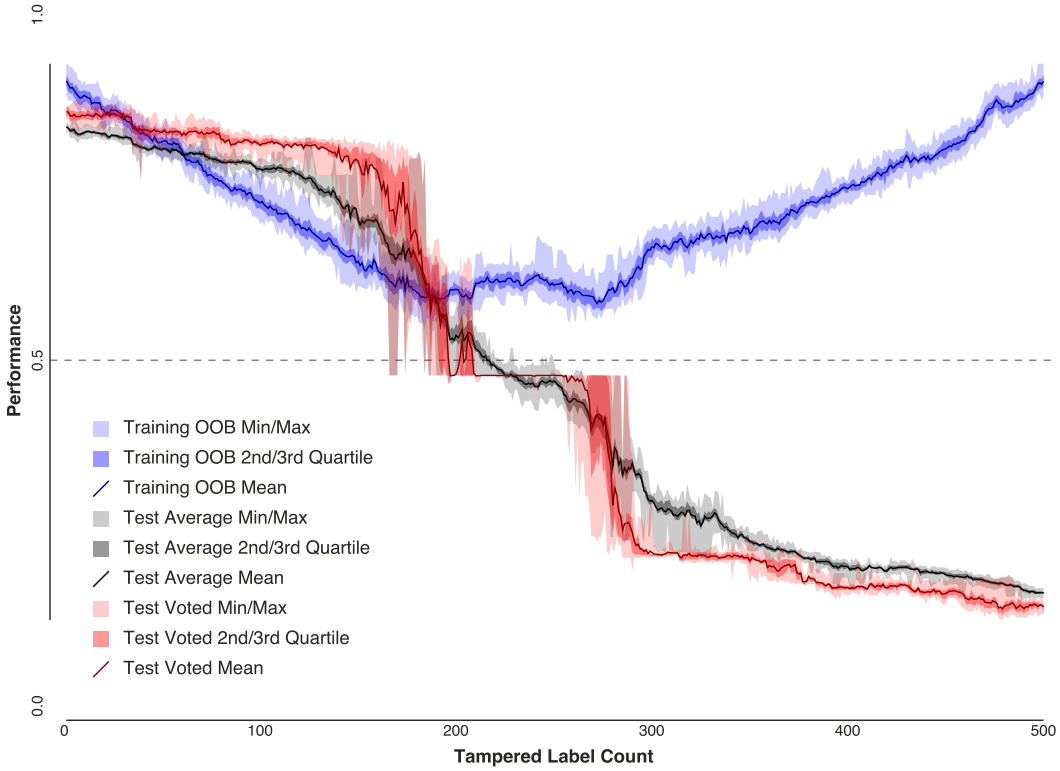


Figure 5: Twenty random defenses against the best random attack on the s500 dataset.

Confidence Attack Metrics

As we began to generate degradation plots for various combinations of data set and attack algorithm (Figure 6), it became clear that there were many potentially conflicting criteria that an adversary might use in choosing an optimal confidence attack:

Efficacy: An adversary might choose one attack over another based on its effectiveness, or how quickly it reduced actual performance – i.e. which attack could achieve a given performance reduction with the lowest cost (least tampering). Attacks that meet this criteria have the steepest slopes for the voted and average curves in Figure 6.

Reliability: An adversary might instead prefer an attack that had low variance, i.e. an attack that doesn't reduce performance as quickly, but instead reduces performance more consistently and predictably, regardless of the parameters used to tune the attack or the defense – thus, the "most reliable" attack. Attacks that meet this criteria have lower variance (narrower envelopes) in Figure 6.

Stealth: Finally, an adversary favoring stealth might choose an attack based on its impact on out-of-bag performance. Since out-of-bag accuracy was the only performance measure available

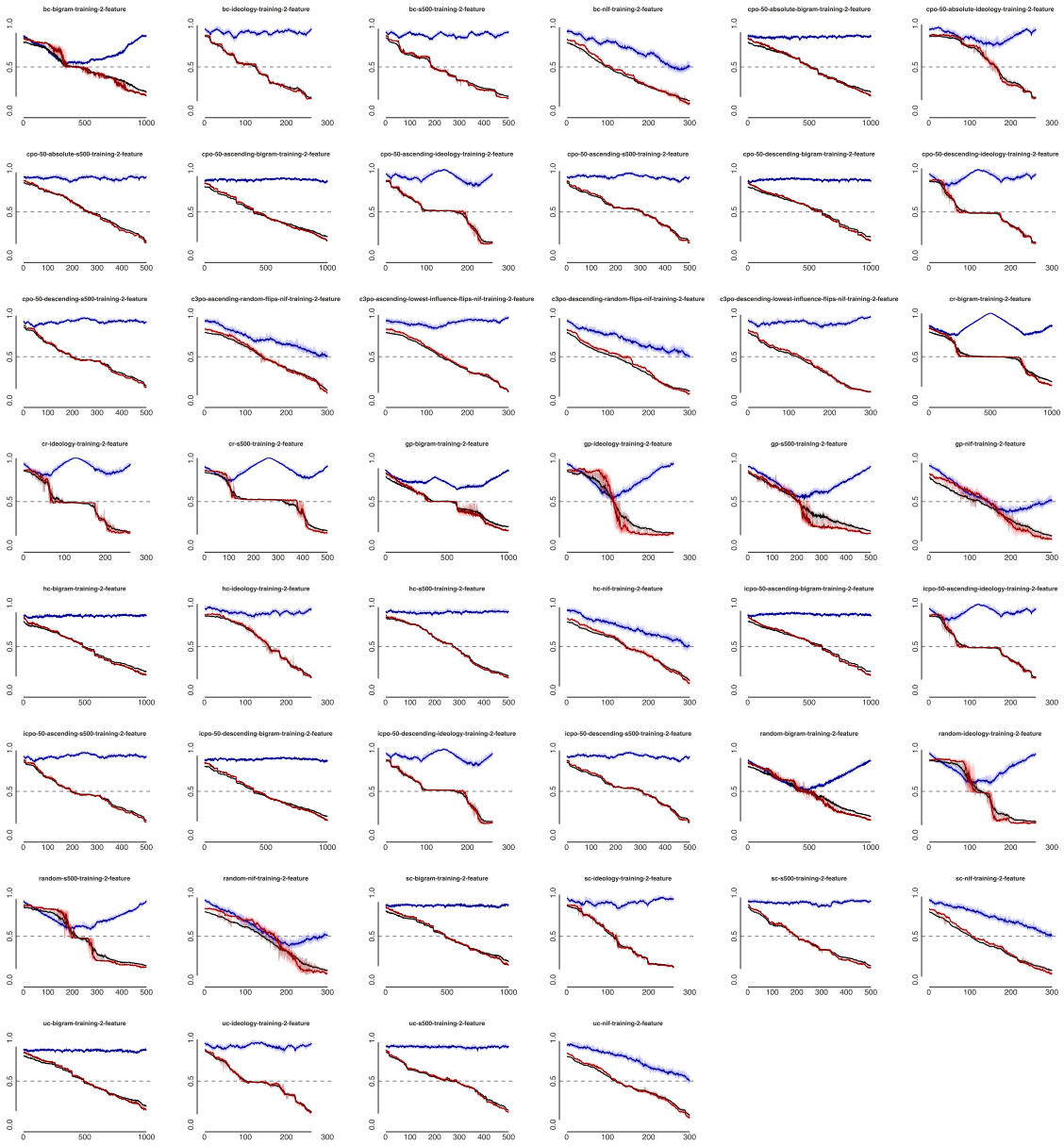


Figure 6: Confidence attack degradation plots for a representative sampling of attack algorithms and datasets.

to the defender in our experiments, an attack that didn't affect out-of-bag accuracy, even if it was less reliable or less effective, would be much less likely to be detected. Attacks that meet this criteria have flat out-of-bag curves in Figure 6.

Degradation plots allowed qualitative comparisons across datasets and algorithms with these three criteria of efficacy, reliability, and stealth in mind, but we wanted to explore more quantitative alternatives. Thus we created several metrics derived from the raw accuracies, including the following:

Absolute Performance Degradation (apd) To score the efficacy of an individual attack, we defined Absolute Performance Degradation to be the signed area of the region bounded by the untampered voted performance v_0 and tampered voted performance v_i of the attack for n label flips:

$$\sum_{i=0}^{n-1} v_i - v_0$$

n could represent the attacker's budget, or for an overall analysis, could represent the full number of data points in the training data. Where the tampered performance is higher, the area is positive; where the tampered performance is lower, the area is negative (Figure 7). From the standpoint of an attacker, lower values are better.

Figure 7: How the absolute performance degradation metric is computed.

For two attacks that otherwise achieve identical performance reductions, the attack that reduces performance sooner and/or more quickly will achieve a better score — APD rewards early, sustained reductions in ensemble performance, which we assume to be the primary goal of an adversary.

Consistency (consistency) To evaluate the reliability of an attack algorithm, we defined the Consistency metric as the area of the region bounded by the minimum voted performance $v_{i_{min}}$

and the maximum voted performance $v_{i_{max}}$ across a collection of attacks for n label flips:

$$\sum_{i=0}^{n-1} v_{i_{max}} - v_{i_{min}}$$

From the standpoint of the attacker, lower values are better, indicating more consistent attack performance for a collection of attacks (or a range of defenses).

Delta Out-of-Bag (doob) To evaluate the "stealthiness" of an attack, we define the Delta Out-of-Bag metric for confidence attacks as the difference between the maximum and minimum values for the out-of-bag accuaracy across an entire attack. From the attacker's perspective lower values are better, suggesting that an attack has a lower impact on the performance that the defender can monitor.

We considered other metrics, and do not consider the list above to be exhaustive. In particular, we investigated means for more strongly weighting metrics when the budget is low, the idea being to reward attacks that are quickly effective without having to separate investigate and report the effectiveness of an attack at 5% tampering, 10% tampering, etc. We did not complete this investigation (mostly due to lack of time; more work would be fruitful here), and so do not have a metric that captures, for instance, our qualitative judgment that CPO is one of the most effective attacks in that it often has the sharpest initial slope. APD at the semi-arbitrary choice of 10% might capture that fact, but overall APD does not, as CPO is no more effective at higher budgets than the other attacks.

Confidence Attack Analysis

Those caveats in mind, Figure 6 and Tables 3, 4 and 5 summarize our metrics for a representative subset of the confidence attack experiments that we ran. One of our first general observations was that the results tend to be very dataset-specific, which is why the tables have been grouped by dataset. We note:

- For all of the datasets tested, there were significant differences in APD among algorithms, meaning that the choice of algorithm is important for an attacker wishing to reduce performance quickly.
- Within each dataset, differences among consistency scores were so low as to be practically nonexistent. So while an adversary may value consistency in an attack, for a given dataset the attack algorithms we tested had little affect on consistency.
- The differences among DOOB scores for each dataset in our experiments were dramatic, confirming that the choice of attack algorithm can dramatically affect the attacker's likelihood of detection.

- For all of our two-class datasets, Greedy Pessimal (gp) had the lowest APD (reduced performance the fastest). However, in all of the data sets it had the highest values of DOOB, which means that a Greedy Pessimal attack is relatively easy to spot via cross-validation self-assessment.
- Brute Clustering (bc) and Subtle Clustering (sc) were in the top three algorithms for APD in three out of four datasets.
- Understated Cluster (uc), Subtle Clustering (sc), and Heterogeneous Clustering (hc) are all top-four algorithms for DOOB in three out of four datasets.
- Therefore Subtle Clustering (sc) stands out as being both relatively effective *and* relatively subtle and hard to spot via the cross-validation metric.
- The three-class nif dataset behaved differently from the other two-class datasets along all dimensions, suggesting that modeling the correct number of classes in a dataset is a critical consideration in choosing attacks.

4.4 Evasion Attacks

Due to time constraints we were only able to explore two types of evasion attack before the end of the project. Like confidence attacks, evasion attack algorithms were responsible for identifying a set of training observation labels to be flipped, an ordering, and the corresponding target labels. However unlike confidence attacks, an evasion attack is generated for a specific test observation that the adversary wishes to have misclassified, which we refer to as the "evader".

Nearest Neighbor (nn): The Nearest Neighbor attack computes distances between the evader (test observation) and every observation in the training data, using a Euclidean distance measure and a subset of features identified as statistically important. Training observation labels are then flipped in-order based on their distance from the evader, from closest to furthest away.

Nearest Neighbor Cohort (nnc): The Nearest Neighbor Cohort attack is computed similarly to Nearest Neighbor, except that only training observations with the same ground truth label as the evader (its cohort) are flipped.

4.5 Evasion Attack Experiments

Our evasion attack experiments were implemented using the same software stack as our confidence attacks (Section 4.3), and followed a similar process:

1. Choose a dataset. Split the dataset into training and test observations, using random sampling without replacement.

Experiment	mean(apd)	std(apd)
gp-b1000-training-2-feature	-357	10.1
icpo-50-descending-b1000-training-2-feature	-352	8.6
cpo-50-ascending-b1000-training-2-feature	-352	8.56
random-b1000-training-2-feature	-349	8.58
hc-b1000-training-2-feature	-342	9.35
bc-b1000-training-2-feature	-335	8.62
cr-b1000-training-2-feature	-333	8.43
cpo-50-absolute-b1000-training-2-feature	-327	9.9
sc-b1000-training-2-feature	-324	8.82
uc-b1000-training-2-feature	-320	8.68
cpo-50-descending-b1000-training-2-feature	-307	9.12
icpo-50-ascending-b1000-training-2-feature	-307	9.1
gp-ideology-training-2-feature	-108	3.6
sc-ideology-training-2-feature	-104	2.51
bc-ideology-training-2-feature	-103	2.33
cr-ideology-training-2-feature	-101	2.03
random-ideology-training-2-feature	-97.8	2.87
cpo-50-descending-ideology-training-2-feature	-95.4	2.27
icpo-50-ascending-ideology-training-2-feature	-95.4	2.28
uc-ideology-training-2-feature	-93.3	2.11
icpo-50-descending-ideology-training-2-feature	-91.9	2.35
cpo-50-ascending-ideology-training-2-feature	-91.9	2.36
hc-ideology-training-2-feature	-75.2	2.63
cpo-50-absolute-ideology-training-2-feature	-72.8	2.42
sc-nif-training-2-feature	-120	3.15
c3po-descending-lowest-influence-flips-nif-training-2-feature	-119	2.74
bc-nif-training-2-feature	-118	3.01
c3po-descending-random-flips-nif-training-2-feature	-115	3
gp-nif-training-2-feature	-114	3.45
uc-nif-training-2-feature	-107	3.26
random-nif-training-2-feature	-101	3.68
c3po-ascending-lowest-influence-flips-nif-training-2-feature	-98.6	2.84
hc-nif-training-2-feature	-98.4	3.14
c3po-ascending-random-flips-nif-training-2-feature	-97.4	3.01
gp-s500-training-2-feature	-225	4.45
bc-s500-training-2-feature	-205	2.97
sc-s500-training-2-feature	-205	2.79
random-s500-training-2-feature	-194	3.53
icpo-50-ascending-s500-training-2-feature	-190	3.16
cpo-50-descending-s500-training-2-feature	-189	3.19
uc-s500-training-2-feature	-180	3.7
hc-s500-training-2-feature	-174	2.78
cr-s500-training-2-feature	-173	3.31
cpo-50-ascending-s500-training-2-feature	-170	3.67
icpo-50-descending-s500-training-2-feature	-170	3.7
cpo-50-absolute-s500-training-2-feature	-168	3.28

Table 3: Absolute Performance Degradation (apd) for a representative sampling of experiments, grouped by dataset and sorted by mean(apd).

Experiment	mean(consistency)	std(consistency)
cpo-50-absolute-b1000-training-2-feature	0.672	0.00794
uc-b1000-training-2-feature	0.672	0.00984
cr-b1000-training-2-feature	0.672	0.009
hc-b1000-training-2-feature	0.672	0.0086
cpo-50-ascending-b1000-training-2-feature	0.672	0.00813
cpo-50-descending-b1000-training-2-feature	0.673	0.00725
bc-b1000-training-2-feature	0.673	0.00833
icpo-50-descending-b1000-training-2-feature	0.674	0.0075
icpo-50-ascending-b1000-training-2-feature	0.674	0.0064
sc-b1000-training-2-feature	0.674	0.00943
random-b1000-training-2-feature	0.675	0.00752
gp-b1000-training-2-feature	0.679	0.00672
uc-ideology-training-2-feature	0.733	0.0125
sc-ideology-training-2-feature	0.735	0.0078
bc-ideology-training-2-feature	0.744	0.0107
icpo-50-descending-ideology-training-2-feature	0.746	0.00887
cpo-50-ascending-ideology-training-2-feature	0.746	0.00887
icpo-50-ascending-ideology-training-2-feature	0.748	0.00971
cpo-50-descending-ideology-training-2-feature	0.748	0.00989
hc-ideology-training-2-feature	0.75	0.00876
cpo-50-absolute-ideology-training-2-feature	0.752	0.00762
cr-ideology-training-2-feature	0.755	0.00848
random-ideology-training-2-feature	0.767	0.00726
gp-ideology-training-2-feature	0.794	0.00609
c3po-descending-lowest-influence-flips-nif-training-2-feature	0.728	0.0109
c3po-ascending-lowest-influence-flips-nif-training-2-feature	0.735	0.0108
c3po-ascending-random-flips-nif-training-2-feature	0.748	0.0132
uc-nif-training-2-feature	0.755	0.0134
c3po-descending-random-flips-nif-training-2-feature	0.757	0.0131
hc-nif-training-2-feature	0.762	0.0116
sc-nif-training-2-feature	0.762	0.0136
bc-nif-training-2-feature	0.763	0.0126
gp-nif-training-2-feature	0.767	0.0125
random-nif-training-2-feature	0.771	0.00756
icpo-50-ascending-s500-training-2-feature	0.721	0.0109
cpo-50-descending-s500-training-2-feature	0.721	0.0118
uc-s500-training-2-feature	0.721	0.0121
cpo-50-absolute-s500-training-2-feature	0.722	0.0109
icpo-50-descending-s500-training-2-feature	0.724	0.0122
cpo-50-ascending-s500-training-2-feature	0.724	0.012
hc-s500-training-2-feature	0.727	0.0101
bc-s500-training-2-feature	0.731	0.0113
sc-s500-training-2-feature	0.735	0.007
cr-s500-training-2-feature	0.736	0.00988
random-s500-training-2-feature	0.736	0.00723
gp-s500-training-2-feature	0.738	0.0104

Table 4: Consistency metric for a representative sampling of all experiments, grouped by dataset and sorted by mean(consistency).

Experiment	mean(doob)	std(doob)
uc-b1000-training-2-feature	0.0697	0.00948
sc-b1000-training-2-feature	0.0706	0.00777
hc-b1000-training-2-feature	0.0732	0.00912
icpo-50-descending-b1000-training-2-feature	0.0753	0.00712
icpo-50-ascending-b1000-training-2-feature	0.0756	0.00769
cpo-50-ascending-b1000-training-2-feature	0.0782	0.00754
cpo-50-descending-b1000-training-2-feature	0.0795	0.00838
cpo-50-absolute-b1000-training-2-feature	0.0838	0.00832
gp-b1000-training-2-feature	0.245	0.00688
cr-b1000-training-2-feature	0.271	0.00556
bc-b1000-training-2-feature	0.366	0.0117
random-b1000-training-2-feature	0.369	0.00968
uc-ideology-training-2-feature	0.134	0.0145
hc-ideology-training-2-feature	0.142	0.015
bc-ideology-training-2-feature	0.143	0.0211
sc-ideology-training-2-feature	0.154	0.0148
cpo-50-descending-ideology-training-2-feature	0.215	0.0105
icpo-50-ascending-ideology-training-2-feature	0.215	0.0105
cpo-50-ascending-ideology-training-2-feature	0.217	0.011
icpo-50-descending-ideology-training-2-feature	0.217	0.011
cr-ideology-training-2-feature	0.227	0.0121
cpo-50-absolute-ideology-training-2-feature	0.251	0.0141
random-ideology-training-2-feature	0.386	0.0191
gp-ideology-training-2-feature	0.442	0.0148
c3po-descending-lowest-influence-flips-nif-training-2-feature	0.151	0.0143
c3po-ascending-lowest-influence-flips-nif-training-2-feature	0.154	0.0101
c3po-descending-random-flips-nif-training-2-feature	0.443	0.0203
sc-nif-training-2-feature	0.446	0.0211
uc-nif-training-2-feature	0.448	0.0201
c3po-ascending-random-flips-nif-training-2-feature	0.457	0.0167
hc-nif-training-2-feature	0.465	0.0242
bc-nif-training-2-feature	0.489	0.0182
random-nif-training-2-feature	0.561	0.0157
gp-nif-training-2-feature	0.573	0.017
uc-s500-training-2-feature	0.081	0.0131
hc-s500-training-2-feature	0.0831	0.0129
cpo-50-absolute-s500-training-2-feature	0.103	0.0103
sc-s500-training-2-feature	0.112	0.00944
bc-s500-training-2-feature	0.123	0.00886
cpo-50-ascending-s500-training-2-feature	0.126	0.0115
icpo-50-descending-s500-training-2-feature	0.127	0.0121
cpo-50-descending-s500-training-2-feature	0.128	0.00922
icpo-50-ascending-s500-training-2-feature	0.128	0.00921
cr-s500-training-2-feature	0.283	0.0121
random-s500-training-2-feature	0.349	0.0136
gp-s500-training-2-feature	0.396	0.0136

Table 5: Confidence Delta Out-of-Bag (doob) for a representative sampling of all experiments, grouped by dataset and sorted by mean(doob).

2. Use statistics computed by the Avatar tree_stats post-processor to identify the two most important features in the training data.

3. Identify a target (tampered) label for each observation in the training data. For two-class data, the tampered label for each observation was simply "the other class". For data containing n classes, we chose one of the other $n-1$ classes at random, with uniform probability.

4. Choose an attack algorithm.

5. For each evader (observation) in the test data, create an attack by flipping labels in the training data in the order specified by the attack algorithm. (We use each test sample individually as an evader in order to build a statistical sense of the effect of evasion regardless of where in feature space an adversary decides to evade.) After flipping each label, generate an ensemble of decision trees using Avatar, and collect the out-of-bag, average, and voted accuracy of the ensemble on test data, plus the ensemble's prediction for the evader's class label. Stop flipping labels as soon as the predicted label no longer matches the ground truth label for that evader, and note how many labels were flipped. Repeat the process until flip counts and before-and-after accuracies had been gathered for every test observation.

   Note that in some cases the ensemble misclassified an evader before any tampering had occurred, so it was possible to have a tamper count of zero.

Evasion Attack Metrics

To evaluate the results of our evasion attack experiments, we considered a variety of metrics that could reflect the adversary's desire for stealth. Recall that the goal of an evasion attack is to cause the misclassification of a single "evader" test observation, so as to escape detection. As with confidence attacks, the adversary would want to minimize their cost (i.e. the amount of tampering necessary), but unlike a confidence attack, we assume that the adversary would also want to avoid affecting the overall performance of the defender's model, to further avoid detection. This led to a set of four metrics:

Flip Count (flips): An obvious metric for the cost of an evasion attack is the number of training labels that have to be flipped to cause the misclassification of the evader's label. Although we assume that a real adversary would use evasion attacks for only one or a few evaders, we computed evasion attacks for every test observation in each data set. This allowed us to compute min, mean, and max statistics on the flip count for each observation, providing concrete bounds on the cost of an evasion attack on any observation, along with the expected (average) cost of an evasion attack for any specific observation. Of course, we assume that the adversary wishes to minimize this cost. Note that, because an ensemble is likely to misclassify at least one evader before any tampering has taken place, the minimum flip count for a dataset / algorithm combination is almost always zero.

Collateral Damage (collateral): As part of the measures captured during our evasion attack experiments, we knew how many observations were misclassified by the ensemble both before

and after an evasion attack. This allowed us to compute the number of observations *other than the evader* that were misclassified as a result of the attack. We call this measure "Collateral Damage", since it reflects observations that are unintentionally affected by the attack. Since the point of an evasion attack is to escape detection, we assume that the adversary wishes to minimize this measure. As with the flip count, we compute min, mean, and max statistics on collateral damage to characterize best, expected, and worst case scenarios for the attacker. In some cases, an attack may actually cause the total number of misclassified labels to *decrease*, leading to a negative collateral damage value.

Delta Voted (dvoted) The Delta Voted measure is the difference in voted accuracy for the ensemble as a whole, before and after the attack, and we compute the min, mean, and max of the measure across all attacks to characterize the attacker's best, expected, and worst case scenarios. Recall that voted accuracy is the measure of the actual performance of the ensemble. From the attacker's standpoint, lower values are better. As with the collateral damage measure, some attacks may have the unintended side effect of *increasing* the ensemble's accuracy, in which case the delta voted measure will be negative.

Delta Out-of-Bag (doob) Like the Delta Voted measure, the evasion Delta Out-of-Bag measures the difference in ensemble out-of-bag accuracy before and after an attack. Recall that out-of-bag accuracy is knowable to the defender in the real world, so minimizing any changes to it is particularly important to the attacker, in order to avoid detection. We compute the same set of descriptive statistics on the doob measure, and as before it is possible to have negative doob values if an attack causes the ensemble performance to increase.

Evasion Attack Analysis

Table 6 organizes the flip count metric for all of our evasion attack experiments, grouped by dataset and sorted by the average flip count. Note that the nearest neighbor (nn) attack outperforms the nearest neighbor cohort (nnc) attack in every case, often by more than an order of magnitude, and that the nearest neighbor average flip counts are bad news for defenders, requiring an average of ten-or-fewer flipped labels to cause misclassification of the evader, a surprisingly low cost for the attacker.

Table 7 displays similarly bad news for defenders using the collateral damage metric – the nearest neighbor attack is able to achieve its goal with very little collateral damage (and often "negative" damage, helpfully improving the defender's overall ensemble performance while simultaneously causing the evader to be misclassified!). Once again, the nearest neighbor cohort algorithm has much lower performance (from the standpoint of the attacker).

Tables 8 and 9 display similar trends, with few exceptions. Particularly note that, for all datasets except the 3-class nif dataset, even the maximum Delta Out-of-Bag metrics for nearest neighbor evasion attacks are extremely low, suggesting that the adversary could attack any observation in the test data with a low probability of detection by the defender. Also note that the corresponding maximum Delta Voted metrics are much higher, reinforcing the notion that out-of-bag metrics are optimistic and can be misleading for the defender.

Experiment	min(flips)	mean(flips)	max(flips)
nn-b1000-test-2-feature-evasion	0	8.57	448
nnc-b1000-test-2-feature-evasion	0	179	500
nn-b1000-training-2-feature-evasion	0	8.69	999
nnc-b1000-training-2-feature-evasion	0	181	500
nn-ideology-test-2-feature-evasion	0	6.64	19
nnc-ideology-test-2-feature-evasion	0	19.9	134
nn-ideology-training-2-feature-evasion	0	6.29	60
nnc-ideology-training-2-feature-evasion	0	19.6	134
nn-nif-test-2-feature-evasion	0	8.33	261
nnc-nif-test-2-feature-evasion	0	51.1	105
nn-nif-training-2-feature-evasion	0	10.5	215
nnc-nif-training-2-feature-evasion	0	48.3	103
nn-s500-test-2-feature-evasion	0	8.19	323
nnc-s500-test-2-feature-evasion	0	34.9	261
nn-s500-training-2-feature-evasion	0	6.77	149
nnc-s500-training-2-feature-evasion	0	65	239

Table 6: Flip count summaries across all experiments, grouped by dataset and sorted by mean(flips).

Experiment	min(collateral)	mean(collateral)	max(collateral)
nn-b1000-test-2-feature-evasion	-14	10.5	404
nnc-b1000-test-2-feature-evasion	-14	118	404
nn-b1000-training-2-feature-evasion	-20	7.78	658
nnc-b1000-training-2-feature-evasion	-20	104	357
nn-ideology-test-2-feature-evasion	-10	4.94	28
nnc-ideology-test-2-feature-evasion	-8	18.3	97
nn-ideology-training-2-feature-evasion	-8	3.73	23
nnc-ideology-training-2-feature-evasion	-4	16.2	97
nn-nif-test-2-feature-evasion	-1	7.93	193
nnc-nif-test-2-feature-evasion	-3	21.4	93
nn-nif-training-2-feature-evasion	-3	8.34	147
nnc-nif-training-2-feature-evasion	-3	17.7	60
nn-s500-test-2-feature-evasion	-10	6.27	261
nnc-s500-test-2-feature-evasion	-14	26	250
nn-s500-training-2-feature-evasion	-14	3.98	142
nnc-s500-training-2-feature-evasion	-14	49.3	249

Table 7: Collateral damage summaries across all experiments, grouped by dataset and sorted by mean(collateral).

Experiment	min(dvoted)	mean(dvoted)	max(dvoted)
nn-b1000-test-2-feature-evasion	-0.014	0.0105	0.404
nnc-b1000-test-2-feature-evasion	-0.014	0.118	0.404
nn-b1000-training-2-feature-evasion	-0.02	0.00778	0.658
nnc-b1000-training-2-feature-evasion	-0.02	0.104	0.357
nn-ideology-test-2-feature-evasion	-0.0383	0.0189	0.107
nnc-ideology-test-2-feature-evasion	-0.0307	0.0701	0.372
nn-ideology-training-2-feature-evasion	-0.0307	0.0143	0.0881
nnc-ideology-training-2-feature-evasion	-0.0153	0.0621	0.372
nn-nif-test-2-feature-evasion	-0.00333	0.0264	0.643
nnc-nif-test-2-feature-evasion	-0.01	0.0715	0.31
nn-nif-training-2-feature-evasion	-0.01	0.0278	0.49
nnc-nif-training-2-feature-evasion	-0.01	0.0588	0.2
nn-s500-test-2-feature-evasion	-0.02	0.0125	0.522
nnc-s500-test-2-feature-evasion	-0.028	0.052	0.5
nn-s500-training-2-feature-evasion	-0.028	0.00796	0.284
nnc-s500-training-2-feature-evasion	-0.028	0.0986	0.498

Table 8: Delta voted accuracy summaries across all experiments, grouped by dataset and sorted by mean(dvoted).

Experiment	min(doob)	mean(doob)	max(doob)
nnc-b1000-test-2-feature-evasion	-0.0287	-0.00305	0.0597
nn-b1000-test-2-feature-evasion	-0.0287	-0.000882	0.0343
nnc-b1000-training-2-feature-evasion	-0.0204	0.00122	0.0456
nn-b1000-training-2-feature-evasion	-0.0197	0.00528	0.0385
nn-ideology-test-2-feature-evasion	-0.0409	0.00909	0.064
nnc-ideology-test-2-feature-evasion	-0.0409	0.0495	0.333
nn-ideology-training-2-feature-evasion	-0.0206	0.0251	0.0747
nnc-ideology-training-2-feature-evasion	-0.0206	0.0689	0.363
nn-nif-test-2-feature-evasion	-0.0179	0.0167	0.316
nnc-nif-test-2-feature-evasion	-0.0376	0.146	0.353
nn-nif-training-2-feature-evasion	0	0.0435	0.373
nnc-nif-training-2-feature-evasion	0	0.157	0.384
nn-s500-test-2-feature-evasion	-0.0205	0.00743	0.0423
nnc-s500-test-2-feature-evasion	-0.0212	0.0476	0.348
nn-s500-training-2-feature-evasion	-0.0476	-0.001	0.0172
nnc-s500-training-2-feature-evasion	-0.0278	0.0857	0.356

Table 9: Delta out-of-bag accuracy summaries across all experiments, grouped by dataset and sorted by mean(doob).

Finally, note that it is surprising and not yet well understood that the Nearest Neighbor Cohort attack scheme performed worse, for the attacker, than plain Nearest Neighbor. The original motivation for Nearest Neighbor Cohort was that if you are trying to hide, for instance, an "Ideology" sample, you would do best to change nearby samples from "Ideology" to "None", but to leave samples that were already "None" alone, in order to create an apparently homogeneous regime of "None". This is not what we observed; attacking all nearby samples, regardless of original class, is a superior scheme, even if that creates new nearby "Ideology" samples. Chaos is somehow superior to consistency here, which bears further investigation.

5 Tamper Remediation Via Ensembles of Outlier Measures

5.1 Introduction to Ensembles of Outlier Measures (EOM)

Section 4 described at length the negative affects label tampering can have on a supervised machine learning model. Here we turn to possible remediation, to the attempt to detect and even correct such tampering. We will explain the structure of our core method, "ensembles of outlier measures" (EOM) in some detail, as it's novel and a bit odd. The core idea is a sort of meta-analysis, to use one supervised machine learning model to look for attacks on another supervised machine learning model.

In other words, we will be working with the same base data, but now with two new "meta" classes of interest, **Changed** and **Unchanged**. To help make our experimental procedure concrete, consider Table 10, which presents an expanded version of our ideology data. Each row is one of our samples, O-class is the original, untampered truth label, and the F_i columns are the original attributes used to predict the O-class label; O-class and the F_i are all that would be required in a standard machine learning application.

Index	O_Class	F_1	...	F_n	T_Class	Tampered	O_1	...	O_k
x_1	Ideology	3.1	...	0.3	None	Changed	1.0	...	0.7
x_2	None	1.3	...	0.7	None	Unchanged	0.0	...	0.9
x_3	None	2.2	...	0.4	Ideology	Changed	3.0	...	0.8
x_4	Ideology	6.0	...	0.9	None	Changed	1.0	...	0.3
x_5	Ideology	3.1	...	0.2	Ideology	Unchanged	9.0	...	0.4
x_6	Ideology	9.7	...	0.1	None	Changed	2.0	...	0.6
...	...	...	...	...	...	...	...	...	...
x_1000	None	8.7	...	0.3	None	UnChanged	3.0	...	0.5

Table 10: Data Organization for the Tamper Detection Problem

As we plan to use the same data scheme to investigate tampering, however, we require some extra information. So the T-class column represents the *tampered* version of O-class, and the Tampered column will have value "Changed" when O-class and T-class differ, else "Unchanged". The Tampered value is the one we wish to predict, and to do so we will need some additional attributes, here depicted in the O_i columns (where "O" was selected to suggest "outlier feature", as will be discussed below).

In other words, we'd like to treat the problem of detecting tampering as a supervised machine learning problem. As such, we require *training data* and descriptive *attributes* suitable to tamper detection.

Training Data for Tamper Remediation

Acquiring training data is the easy part. For any given data set we can generate as many label-tampered variants as desired by simply altering labels synthetically. Further, we can use any of the tampering attacks described in Section 4, and at any level of budget in those attacks. The end result in each case will be the column T-class, where we altered some subset of column O-class. That T-class column thus defines the contents of the Tampered column, as just mentioned, and so it is the Tampered column that will be used as "truth" in the model to be built.

Attributes for Tamper Remediation

It would be wonderfully simple if one could use the primary, already existing, attributes of the data (that is, the F_i columns in Table 10) to fuel the *Changed/Unchanged* analysis. However, there's no particular reason to suspect that the individual attribute vectors, selected for their value in detecting ideological text, would be useful for the purpose of detecting tampering, and in fact, testing confirms that they are not.

However, the various samples do not stand alone, they exist in a N-dimensional space, perhaps with some structure. Conceptually one can think of loose clusters of data points, with the clusters likely being somewhat homogeneous in class. The intuition is that that structure, the homogeneity of the neighbor relationships, is altered by label tampering, and so secondary attributes sensitive to those neighborhood relationships might be sufficiently correlated with tampering to fuel a machine learning model.

5.2 The Current Set of Outlier Features

Accordingly, we investigated outlier measures as a way of characterizing points that were dissimilar to their neighbors. Note this dissimilarity might be a function of a point's position in feature space, of its class, or both, and note further that it is the potentially *tampered* class that is considered here. In other words, the outlier measures operate on the F_i and T-class columns of Table 10.

The current set of outlier measures are:

1. *Boosting weight.* Boosting [**?**] is an ensemble machine learning method that iteratively generates data sets for new classifiers by more heavily weighting data that the current ensemble is getting wrong. That is, it tends to focus attention on difficult to characterize outliers, with the degree of that attention captured in the weight assigned to each sample.

   Here we apply boosting to the original data and the primary labels (in this case, Ideology/None) after tampering, and then extract the boost weights of each sample as an outlier feature. The intuition is that the boosting weights will be high for samples with classes unlike their neighbors.

2. *Ensemble confidence mismatch.* Here we build a bagged ensemble of decision trees [**?**], and use out-of-bag validation [**?**] to classify every sample in the training set. Ensemble confidence in class y_i is the percentage of the ensemble that voted for class y_i. If y_t is the true class, then our ensemble confidence mismatch feature for sample x is 1-p(y_t—x).

   In other words, if the ensemble largely votes for the correct class, the mismatch is small. But if the ensemble fails to put much weight on the correct class, which might correlate with an oddball sample, then the mismatch is large.

3. *Label Propagation and Spreading.* Label spreading methods [**?**] propagate labels to their nearest neighbors in an iterative fashion that tries to find a global consensus. "Nearest" is often tricky and unreliable to define; here we use "semantic similarity" as defined by an ensemble of decisions trees[**?**]. Further, our outlier measure is the delta between the original label of a point and the weight of that label after convergence. If there is no change, then the outlier measure is small. If global consensus strongly suggests that the label should change, then the outlier measure will be higher.

4. *KNN weighted voting.* This is simply the percentage of the K nearest neighbors whose label disagrees with the current label.

5. *Local Outlier Factor.* LOF[**?**] finds anomalous data points by measuring the local deviation of a given data point with respect to its k nearest-neighbors.

6. *Local Correlation Integral.* LOCI[**?**], similarly to LOF, finds anomalous data points by measuring the local deviation of a given data point with respect to the neighborhood described by a certain distance metric.

7. *Density Based Spatial Clustering of Applications with Noise (DBSCAN).* DBSCAN [**?**] is a density-based clustering algorithmthat tries to find high density neighborhoods that are separated from each other by low density neighborhoods. DBSCAN can be used to determine outlierness in a variety of ways; by measuring the current point's distance to the nearest core cluster point and by simply determining if the current point was clustered or determined to be unclusterable "noise".

Tamper Detection and Remediation in General

With the preliminaries in place, it is relatively straightforward to explain how we build a model, detect tampering, and correct for it.

1. Generate synthetic tampered data. That is, start with data like columns O-class and F_i of Table 10, select some samples $(x\_i, \ldots x\_m)$ to tamper according to some attack and budget, and create the columns T-class and Tampered to reflect the tampering.

2. Extract new attributes to capture the outlierness of the samples with the tampered labels. That is, use columns F_i and T-class to determine columns O_i.

3. To train the model, feed the Tampered column as truth (as we are trying to learn to predict the Changed/Unchanged) and the F_i and O_i columns as attributes to the Avatar code, which generates a bagged ensemble of decision trees. (The F_i attributes are not, in general, necessarily expected to be predictive, but one can imagine circumstances when they might be. And as ensembles of decision trees are unaffected by spurious features, there is no reason not to include them.)

4. To apply the model, take a *new* dataset consisting of F_i columns and a possibly tampered Class column. Use the F_i and the truth to compute O_i, then feed F_i and O_i to the Avatar model to predict "Changed" or "Unchanged" for every sample in a "Tampered?" column.

5. Finally, remediate. Since here we are working with two-class problems, we simply change the class of every sample where Avatar predicted "Changed", creating a new column, R-class, of remediated class values. (Other remediations are possible and are under investigation. For instance, one could simply remove the suspect samples, the ones with label "Changed", and operate only on the remainder; if effective, this would more conveniently handle multi-class data.)

Consider Table 11 for an illustration of how remediation is applied. Here we have a column labeled "Class" whose values may or may not be reliable, because we suspect they may have been tampered with. We compute the outlier attributes O_i from F_i and Class, feed O_i and F_i to an Avatar-generated model, and for each sample learn whether the model predicts that the Class column was tampered with. In the notional example of Table 11 we see that Avatar assigns class Changed to samples y_2 and y_1000. Therefore the column R-class matches the column Class, except for y_2 and y_1000, where the class has been converted to the other possibility.

Index	Class	F_1	...	F_n	O_1	...	O_k	Tampered?	R_Class
y_1	Ideology	4.1	...	0.2	3.0	...	0.0	Unchanged	Ideology
y_2	None	2.3	...	0.9	2.0	...	0.2	Changed	Ideology
y_3	None	3.2	...	0.6	5.0	...	0.1	Unchanged	None
y_4	Ideology	8.0	...	0.1	3.0	...	0.6	Unchanged	Ideology
y_5	Ideology	4.1	...	0.4	1.0	...	0.7	Unchanged	Ideology
y_6	Ideology	7.7	...	0.3	5.0	...	0.9	Unchanged	Ideology
...	...	...	...	...	...	...	...	...	...
y_1000	Ideology	8.7	...	0.5	6.0	...	0.8	Changed	None

Table 11: A Notional Illustration of Remediation

6 Tamper Remediation for General Supervised Learning

The previous section described the algorithmic idea, and the core machine learning model structure, behind the use of ensembles of outlier models (EOM) for tamper detection.

Here we will first describe how those ideas were implemented to permit exploratory analysis, indicate how that implementation permitted extensive, even exhaustive investigation of attack and budget trade-offs, and then focus on some specific scenarios to illustrate the results of those experiments.

6.1 Matched, and Mismatched, Attack and Defense Models

As was mentioned in Section 5.1, given the existence of a pristine data set we can generate a synthetic tampered data set by choosing a) any of the tampering attacks described in Section 4 and b) any level of budget in that attack, and then changing the indicated number of labels in the indicated order.

An obvious question arises: *which* attack to choose, and at which budget? If you are a defender, and you know what specific attack the attacker will use, then of course you would train your defense on that attack. Similarly, if you are the attacker and you know which defense is in play, you'll choose the attack most effective against that defense.

Of course, the opponent's choice of strategy will rarely be known. Accordingly, in CADA we nearly exhaustively investigated *all* pairs of attacks and defenses, across the full range of possible budgets. This allowed us to build intuition in a complex and novel space, and set us up to investigate robustness questions, such as: if I'm the defender, which assumption about the attacker gives me the *best* worst case outcome, even if I'm wrong?

6.2 The Experimental Set-Up

Since the experimental design (and necessarily, its implementation) is therefore fairly complex, and often confused us as we built it, we here revisit the algorithm outline of Section 5.2, filling in the additional scaffolding and detail necessary for implementation. As with the attack implementations, the tamper detection and remediation ideas were implemented in Python, making extensive use of NumPy and SciPy for computation. The experimental interface was implemented in IPython, for ease of visualization and interactive control. See Appendix A for further discussion and pointers.

So, the implementation details:

1. Select one of the datasets analyzed in Section 4.3. The selected data will thus include a "tamper training" dataset ("ttrain" for the purposes of this process description) and a "tamper

test" dataset ("ttest"), each composed of truth labels, features, and a series of corresponding pre-calculated attacks for the dataset. (Recall that in its final form, a a single specific attack is just an ordering of the samples in a data set, indicating the order in which one would choose to change their labels.)

2. Identify a budget mode (count or percent), budget range and granularity (increment) for ttrain. Designate the truth column and feature columns for ttrain.

3. Identify a budget mode (count or percent), budget range and granularity (increment) for ttest. Designate the truth and feature columns for ttest.

4. Iterate over all of the pre-calculated attack algorithm combinations for the ttrain and ttest datasets. Note that ttrain and tttest attacks can be different, as well as the ttrain and ttest attack budgets.

5. For the current attack algorithm, increment over the attack budget levels on the ttrain dataset.

   (a) Flip labels (for two-class data simply switch an observed label to "the other class") in the ttrain dataset according to the ttrain attack budget. This supplements the original data file by generating 1) a "changed" column that designates whether a label was flipped or not and 2) a "flipped" column that holds the value of the tampered labels. (To re-emphasize this point: the "flipped" column has a label for all samples, but only some of them may have been tampered with. For example, if the current attack budget level is 10%, then 90% of the "flipped" column values will still be the same as the original truth values.)

   (b) Extract the outlier features as described in Section 5.2 from the ttrain features, based on the *tampered* ttrain labels.

   (c) Use Avatar to build a *tamper* detection decision ensemble, using the calculated ttrain outlier features (optionally supplemented by the original ttrain features) as features and the "changed" ttrain column as truth.

   So again note that the resultant Avatar model will learn to predict "Changed" or "Unchanged", not "Ideology/None" or whatever the original task might have been, and so will eventually be evaluated as to how well it can detect tampered data in a test set.

   (d) Once the Avatar tamper detection model is built, use the tree_stats code, Section 2, to determine and record the relative importance of each of its features, for later analysis.

   (e) Use Avatar to build a *classification* decision ensemble, using only the original ttrain features as features and the tampered ttrain labels (that is, the "flipped" column) as truth. The resulting model thus predicts "Ideology/None" or whatever the original task might have been. Testing it will tell us what accuracy loss was induced by the tampering. We wished to record this in pursuit of the idea that certain outlier features might be more or less useful accordingly to how severely the data had been tampered with.

6. Given the current ttrain attack budget increment, increment over the ttest attack budget levels to create testing data.

(a) Flip labels (for two-class data simply switch an observed label to "the other class") in the ttest dataset according to the ttest attack budget, again generating 1) a "changed" column that designates whether a label was flipped or not and 2) a "flipped" column that holds the value of the tampered labels.

(b) Extract the outlier features as described in Section 5.2 from the ttest features based off the tampered ttest labels.

(c) Using the tamper detection decision ensemble developed from the ttrain attack budget, analyze the ttest outlier features and generate 1) a "predicted" column that designates whether the model expects that a label has been tampered with and 2) a 'Overall Voted Accuracy" which describes how well a tamper detection model performs on tampered data.

(d) Use Avatar to build a classification decision ensemble, using the ttest features as features and the tampered ttest labels as truth. Test this model on the *untampered* ttrain dataset, calculating an "Overall Voted Accuracy". The point here is to determine how well a tampered classification model performs on untampered test data, by way of comparison with subsequent remediation.

(e) Using the predicted ttest labels, remediate the tampered ttest data (for two-class data simply switch an observed label to "the other class") generating a new "remediated truth" column.

(f) Use Avatar to build a classification decision ensemble, using the ttest features as features and the remediated truth ttest label as truth. Test the built model on the untampered ttrain dataset, calculating the repaired "Overall Voted Accuracy" to describe how well a remediated classification model performs on untampered data.

In sum, for a given data set we exhaustively vary four parameters:

- presumed attack strategy (that is, the strategy the defender simulates to built their tamper detection model),
- presumed attack budget,
- actual attack strategy,
- actual attack budget.

An example setting of that 4-tuple might be (brute-clustering, 10%, random, 25%). This is interpreted as "the defender builds a tampering detection model presuming that the attacker will corrupt the top 10% of the labels in the order suggested by brute clustering, but in fact the attacker corrupts the 25% of the labels in a random order."

For each of the 4-tuples investigated, we record:

- The accuracy of the defender's ability to detect tampered points.

- The relative importance of each of the outlier features in detecting that tampering. (Technically, this depends only on the first two values of the 4-tuple).
- The baseline accuracy of the classification model if we don't remediate.
- The accuracy of the classification model if we use the tamper detection model to first remediate the labels detected as tampered.

6.3 Example Experimental Results

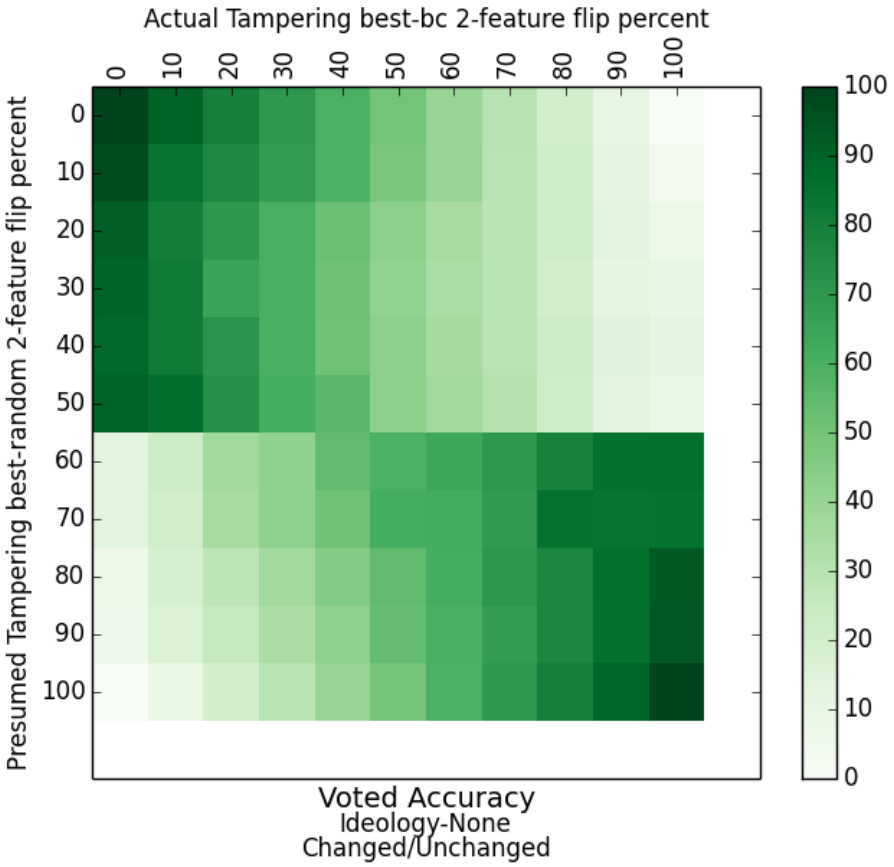


Figure 8: Tamper detection accuracy for brute-clustering tampered data presumed to be randomly tampered.

The next challenge was how to best to investigate and absorb all the experimental data thus generated on the the effects of varying combinations of attack and defense mechanisms. We thus created and investigated a number of summary visualization methods.

One of these was simple accuracy heatmaps, in which the x and y axis indicate some specific choice for presumed and actual attack strategies, and the x and y position indicate the attack budget. See Figure 8 for an example tamper detection heatmap. Some notes on interpretation:

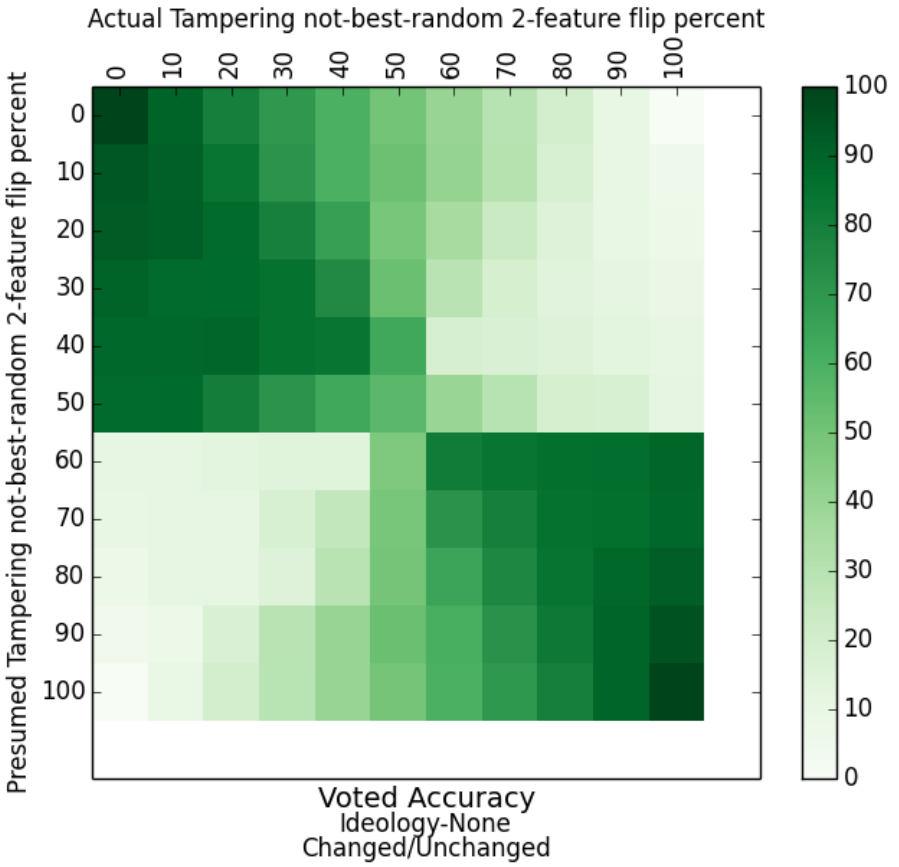


Figure 9: Tamper detection accuracy for randomly tampered data presumed to be randomly tampered.

- The top left cell is where no tampering actually occurs and none is presumed. Thus the trivial "no tampering exists" model is perfectly accurate.
- That also explains what might seem to be a puzzling increase in accuracy at the bottom right. Here nearly all data points are corrupted (at x near 100%), but since the defense model is expecting this (y near 100%), the "all points are tampered" model is similarly almost perfectly accurate.
- To interpret, say, the second row, y=10, this indicates the defenders gradually degrading ability to spot tampering when the defender assumes 10% tampering, but the level of tampering actually increases.
- The second column, x=10, indicates the results where the level of tampering is actually 10%, but the defender assumes increased levels of tampering. The cliff at 50% indicates the unexpected effect (replicated widely) that as long as the defender is not *too* paranoid, its tamper detection accuracy is fairly robust even though it does not precisely guess the attackers budget.
- Finally, compare Figure 8, where there is a mismatch between attacker and defense strategy,

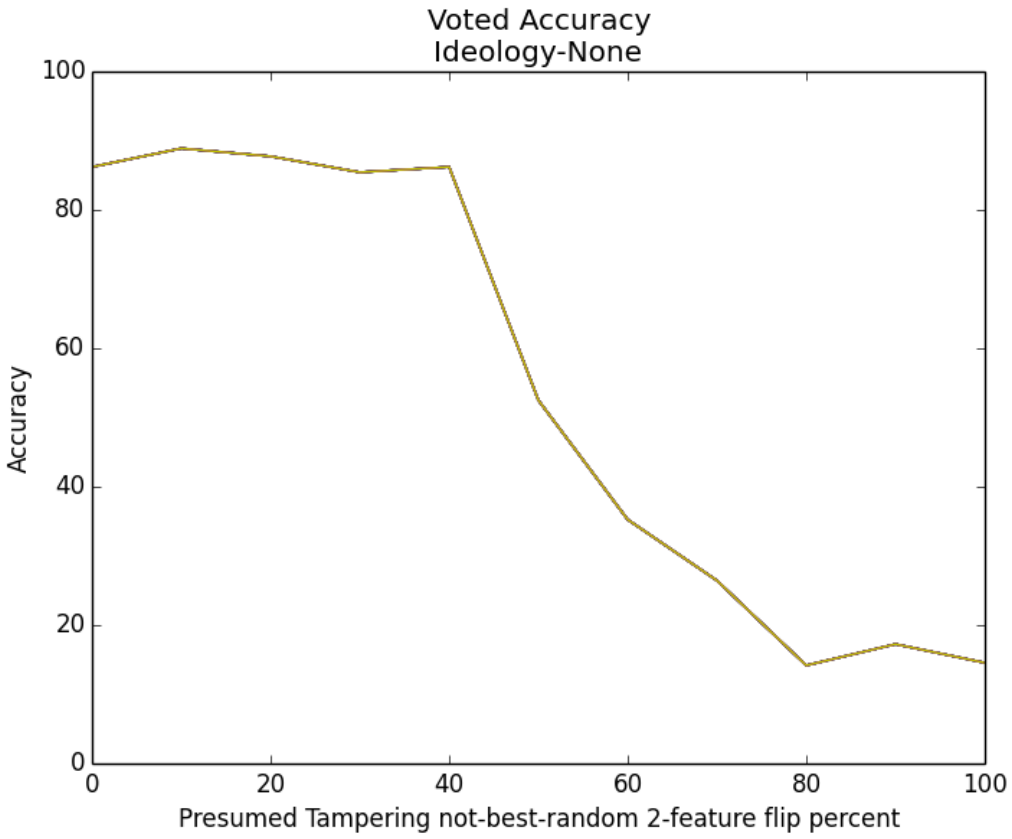


Figure 10: Classification accuracy of randomly tampered model.

with Figure 9, where the defender accurately guessed the attacker strategy. The heatmaps are largely similar, generating the unexpected observation that getting the presumed attack wrong did not materially reduce tamper detection accuracy; getting the budget right is a much more important consideration.

As a second visualization mechanism, we generated simple accuracy plots showing the degradation of classification accuracy as the amount of tampering increases. These basically replicate the degradation plots of Section 4.3, but for the specific attacks generated, and help to serve as a baseline comparison to remediation plots. One example is in Figure 10; the fact that accuracy stays close to constant until roughly 40% tampering is reached is a tribute to the robustness of ensembles of decision trees . . . to *random* label noise. This was initially encouraging, but contrast Figure 11, where the accuracy degradation is due to the "subtle clustering" attack, and is nearly linear with the attack budget. And indeed, as was earlier illustrated in Figure 6, almost all of the non-random attacks lack the robustness to noise illustrated in Figure 10.

To investigate the effectiveness of remediation, we first returned to heatmaps such as Figure 12. These heatmaps describe the accuracy of a remediated classification model in classifying tampered data. However, we realized that in and of themselves they can be deceptive. Figure 12 suggests

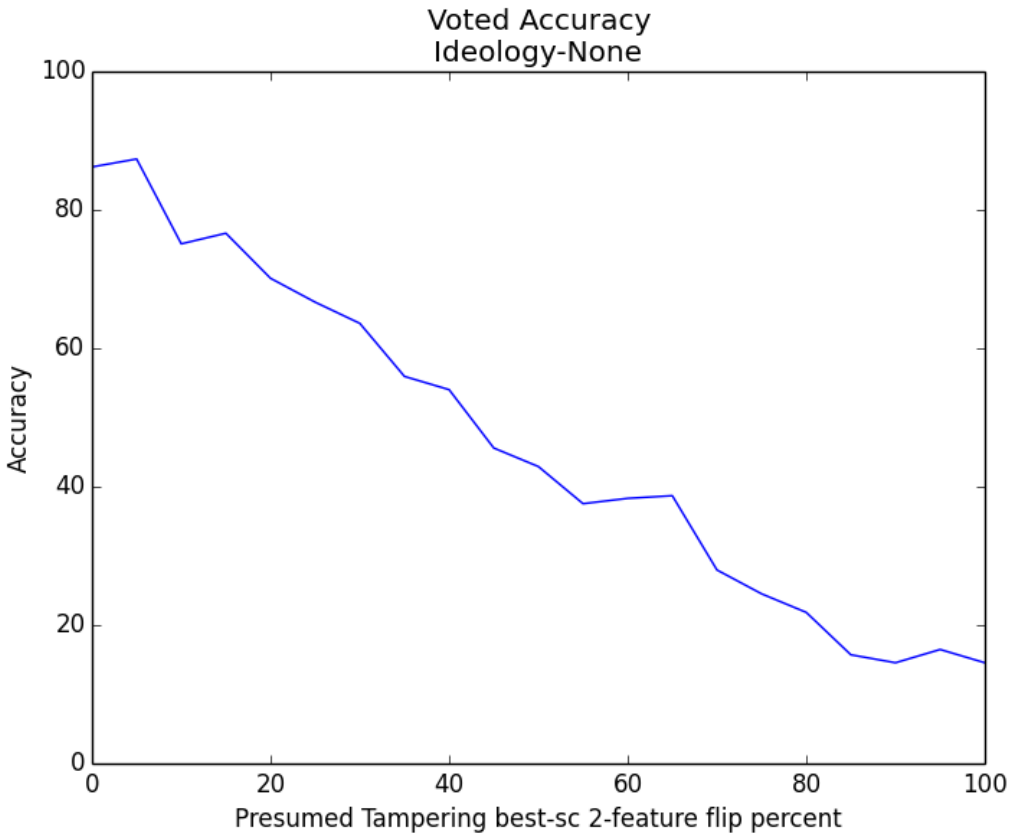


Figure 11: Classification accuracy of subtle-clustering tampered model.

that post-remediation accuracy can be pretty good; but recall the first section of Figure 10, which indicated that *pre*-remediation accuracy was also not bad. Accordingly we also compute "delta accuracy" heatmaps, depicting the change in classification accuracy due to remediation. An example is in Figure 13. Some notes on interpretation:

- For this strategy trade-off (presume a random attack, and the actual attack is indeed random), presuming a budget of less than 50% doesn't materially help or hurt. It consistently only helps if the attack budget really is 50%.

- Presuming an attack above 50% materially hurts if the attack is actually less than 50%, but materially helps if that presumption is correct.

These remediation heatmaps are individually interesting, but they also made it difficult to pick out trends. The ultimate goal of a defender here would be to look at *all* of these heatmaps and try to determine what specific choice of defense strategy and budget she should make. So we created a number of graphics intended to illustrate various optimistic and pessimistic scenarios resulting from a specific defense choice.

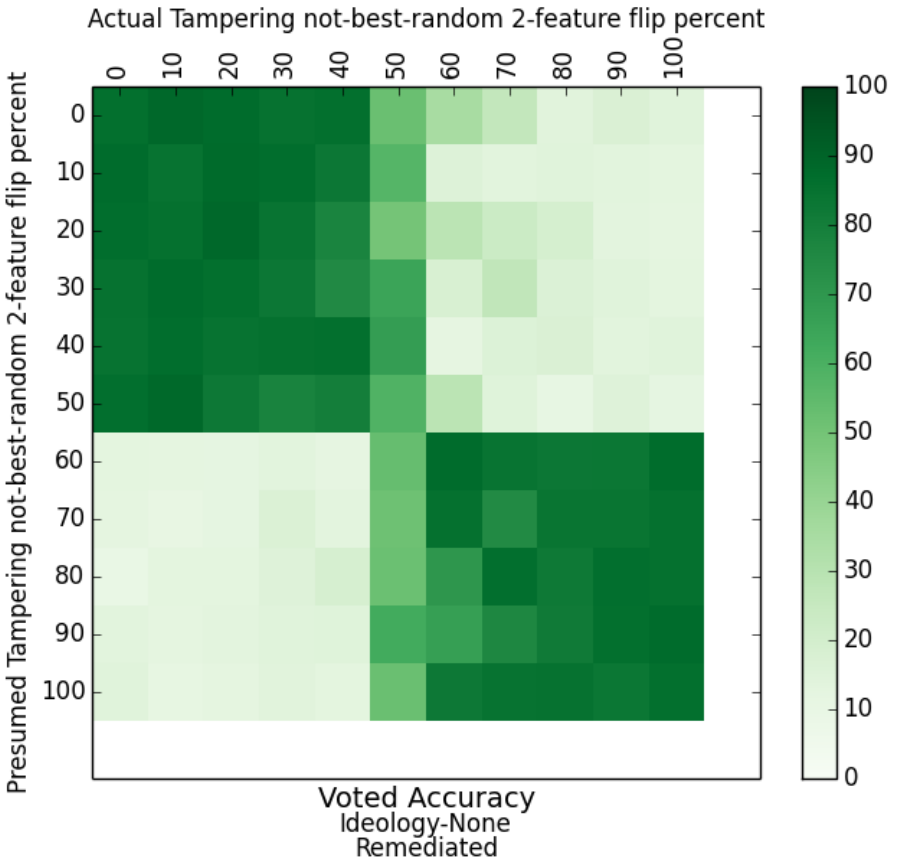


Figure 12: Classification accuracy of remediated model of randomly tampered data presumed to be randomly tampered.

So first consider Figure 14a, just to understand the related figures to follow. This figure comes from attacking the ideology data set (see "Ideology/None" in the upper label) with a random attack (see the lower label). We are plotting accuracy on the y-axis, and attack budget on the x-axis. The dotted purple line ("no remediation" in the legend) indicates the accuracy if we don't remediate. The rest of the lines indicate the accuracy if we *do* remediate with models based on the various presumed attacks; these remediation values are a single value from the columns in heatmaps such as Figure 12, with those values selected according to a specific "scenario", to be discussed. In any case, the hope (sometimes to be dashed) would be that the remediated curves would be higher than the no-remediation curve.

The two plots in Figure 14, called the "Maximum" accuracies in the caption, show a best case scenario. Here, for each possible level of attack tampering (the x-axis), we have selected from that column the *maximum* available remediated classification accuracy. Therefore the accuracy curves in this figure assume that the defender was lucky enough to pick the best possible level of tampering for remediation at each attack tampering granularity.

While the results of the that curve appear very promising, of course it is unrealistic to expect a defender to be optimally picking tampering levels for remediation in all cases. Therefore next we

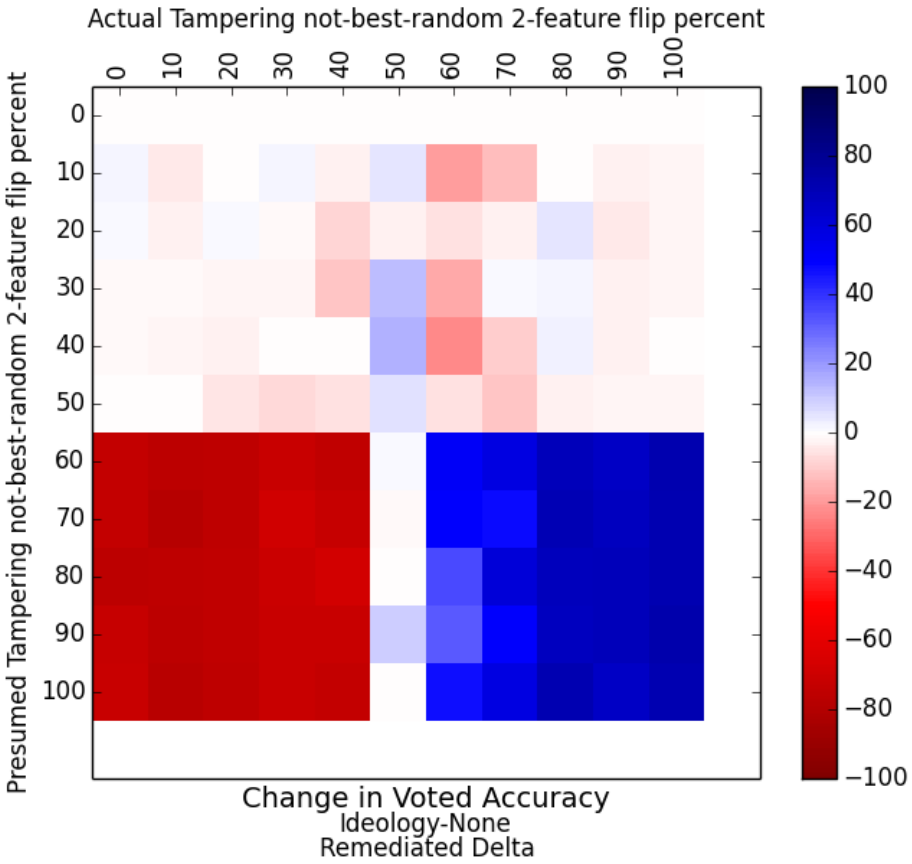


Figure 13: Change in accuracy due to remediated model n of randomly tampered data presumed to be randomly tampered.

looked at the *diagonals* of our remediated classification heatmap curves and plotted them against unremediated accuracy as shown in Figure 15. These curves describe how well remediation works when the defender can guess the exact amount of tampering an attacker has performed on the data. There are a couple of interesting anecdotal points to note here:

- As shown in the Brute Clustering subgraph of Figure 15, when a defender selects a good remediation mechanism (here, best-icpo-50-descending) and correctly guesses the tamper attack budget, remediation can offer some model resilience against an intelligent attack. What's particularly interesting here is that the best attack to presume is *not* the actual attack. Brute clustering is the actual attack, but defending against it specifically is actually one of the worst performing strategies.
- Also note that when the actual attack is random (in the first subfigure), there is little value in *any* of the remediation methods, until and unless the attack budget gets above around 45%. Before that, the natural resilience of ensembles of decision trees is so effective that most remediation methods hurt, even if they correctly guess the attack budget.

For another slice at the data, a "good but not best outcome", assume we will examine heatmaps

such as Figure 12 and find the *row* with the highest summed value, and use that as a fixed presumed tamper budget. This is what we see in Figure 16; the final two-digit value in the legend names indicates the defense tamper budget that was best for each attack. Similarly, Figure 17 indicates a "bad but not worst outcome" where the defender has unluckily selected a defense budget which has the *lowest* summed accuracy across all the attack budgets.

Finally, we note that the plots to date have attempted to capture metrics that indicates the robustness of a presumed defense over the various attacks and budgets the attacker might attempt. We also considered acting on the happier assumption that an attacker has limited resources and therefore will likely have minimal capabilities of tampering with our data. The line graphs shown in Figure 18 give insight into this type of assumed scenario. In order to create this graph, we selected the case where an attacker was tampering with 10% of our data. We then assumed a defender who selected the defense remediation budget that gives the best remediated classification accuracy against the 10% budget attack. We then plotted the remediated classification accuracy at the selected best granularity across all possible attack budgets, in order to assess the impact of the attacker indeed attacking at a level other than 10%. Again (and again, anecdotally), the results seem to favor remediation if an intelligent attack is expected, as overall many defense mechanisms offer resilience at early stages of tampering and do not significantly decrease performance at further levels of tampering. However, again it appears that remediation against an unintelligent attack does not earn a defender any valuable accuracy improvements.

6.4 Conclusions and Next Steps

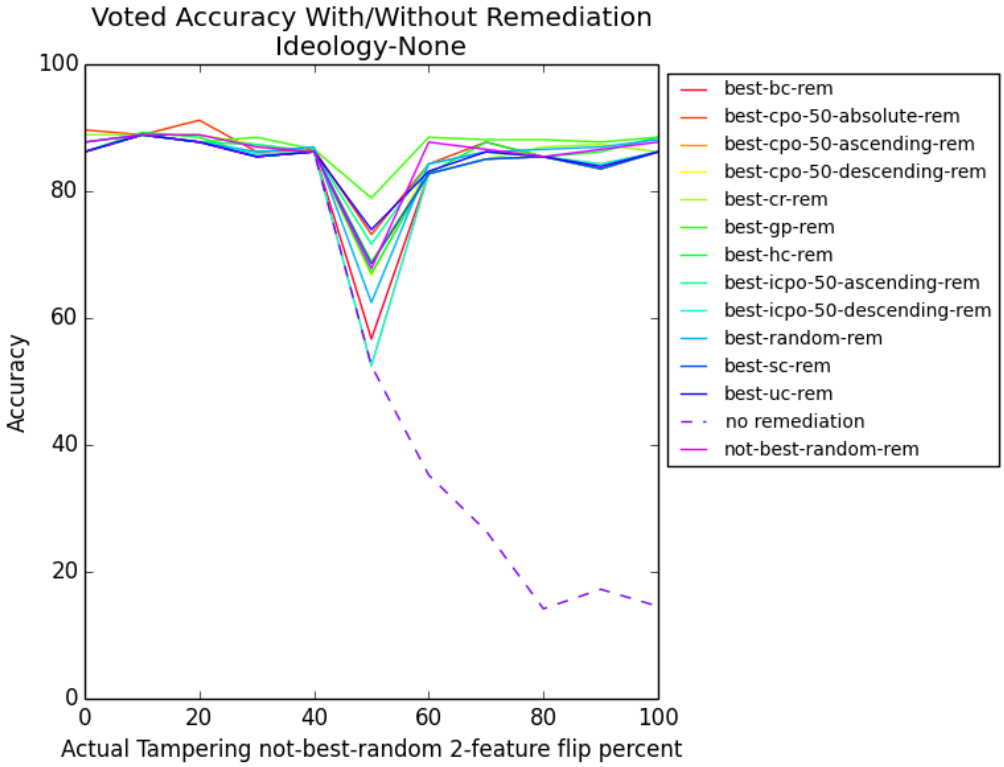
In conclusion, we have:

- Described Ensembles of Outlier Methods (EOM), a method for building a meta-model for detecting tampered data via supervised machine learning.
- We have shown how to use EOM to remediate tampered data by detecting, and correcting, the labels of suspect data points.
- We have implemented all of this in a code base that permits the exhaustive experimental investigation of presuming certain attacks and budgets when the actual attacks and budgets may be different.
- We have designed and implemented a wide variety of plots and metrics to permit investigation of that exhaustive investigation from a variety of perspectives.

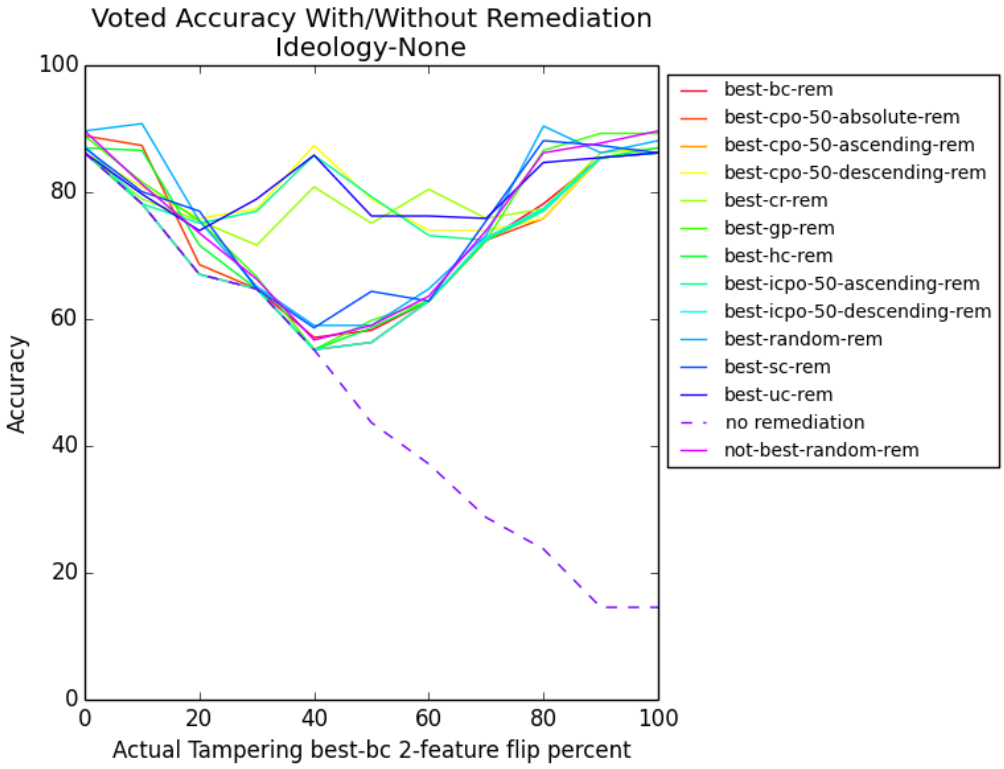
The sensible next steps would be to develop metrics and post-processing tools to illuminate:

- What combination (which 4-tuple) was worst for the defender? What combination was worst for the attacker? What trends are observable?

- Similarly, what presumed attack was most robust for defender? Which actual attack was most robust for attacker?
- Which outlier features were most useful over all?
- Which outlier features were most useful when unremediated accuracy was high? Was low? Is there a difference?
- If we look at the relative efficacy of the outlier features for the 4-tuple where the defender does worst, or where the defender does best, what insight might that give us as to how to improve or add to our current set of outlier features?

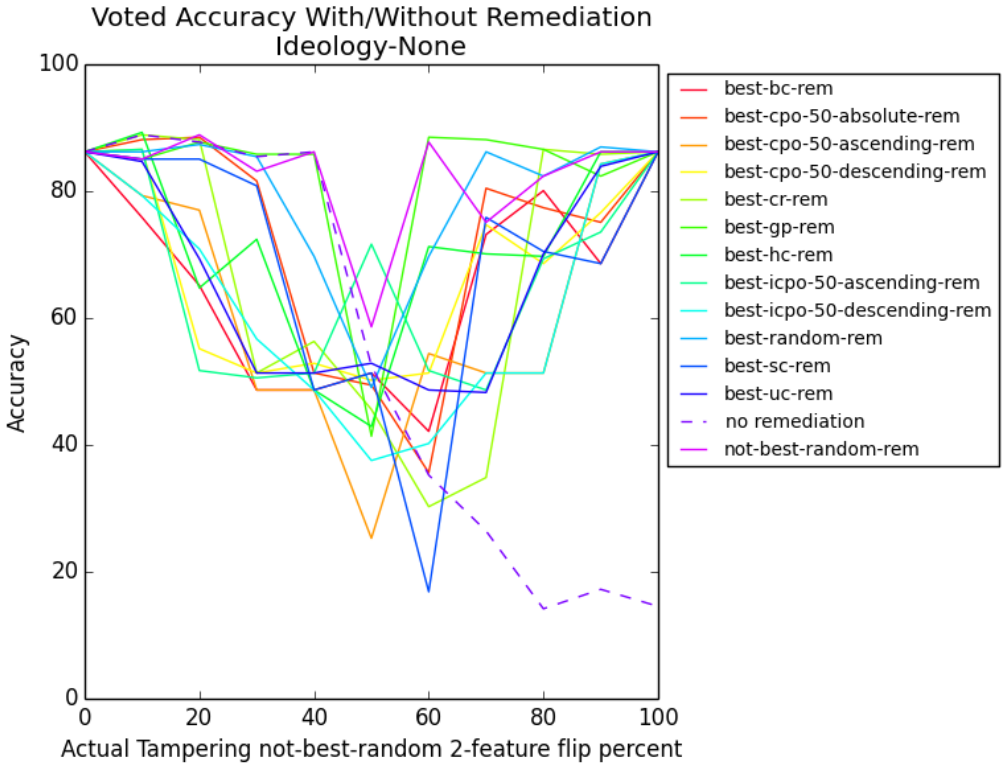


(a) Presumes Random Tampering

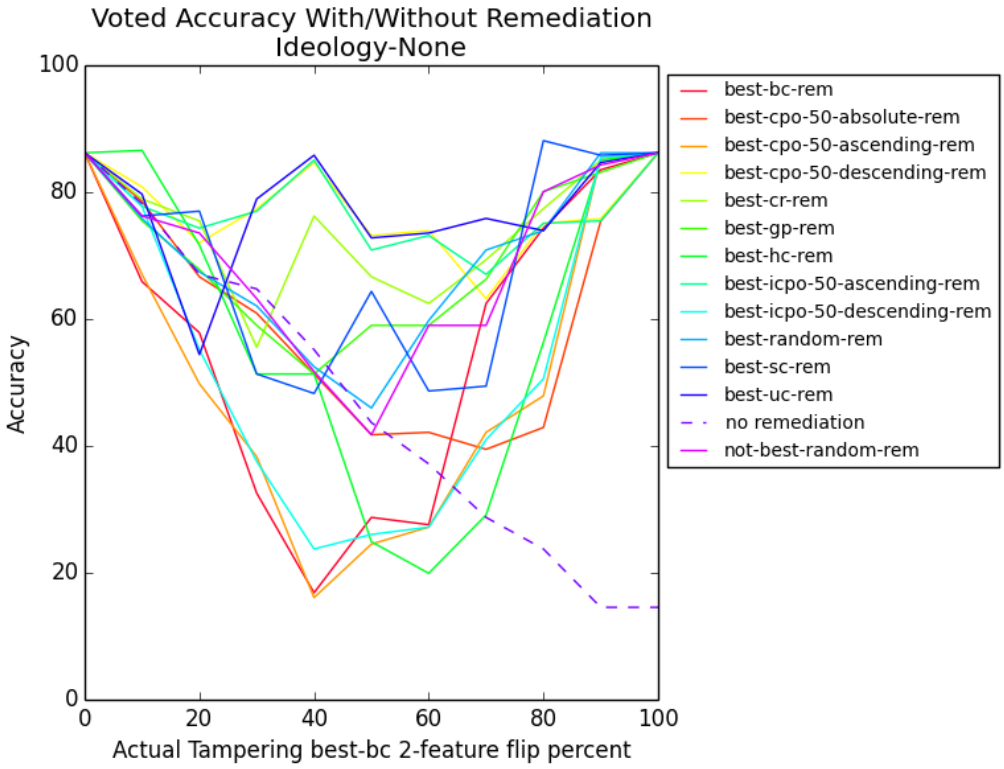


(b) Presumes Brute Clustering Tampering

Figure 14: Remediation Maximum Accuracies

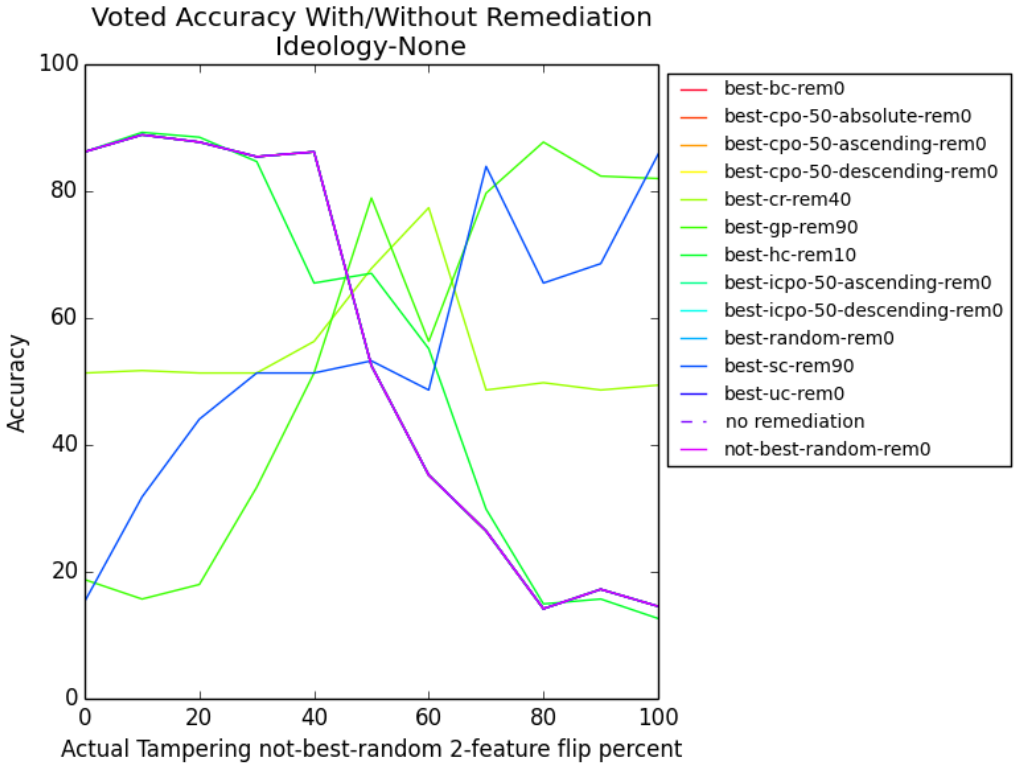


(a) Presumes Random Tampering

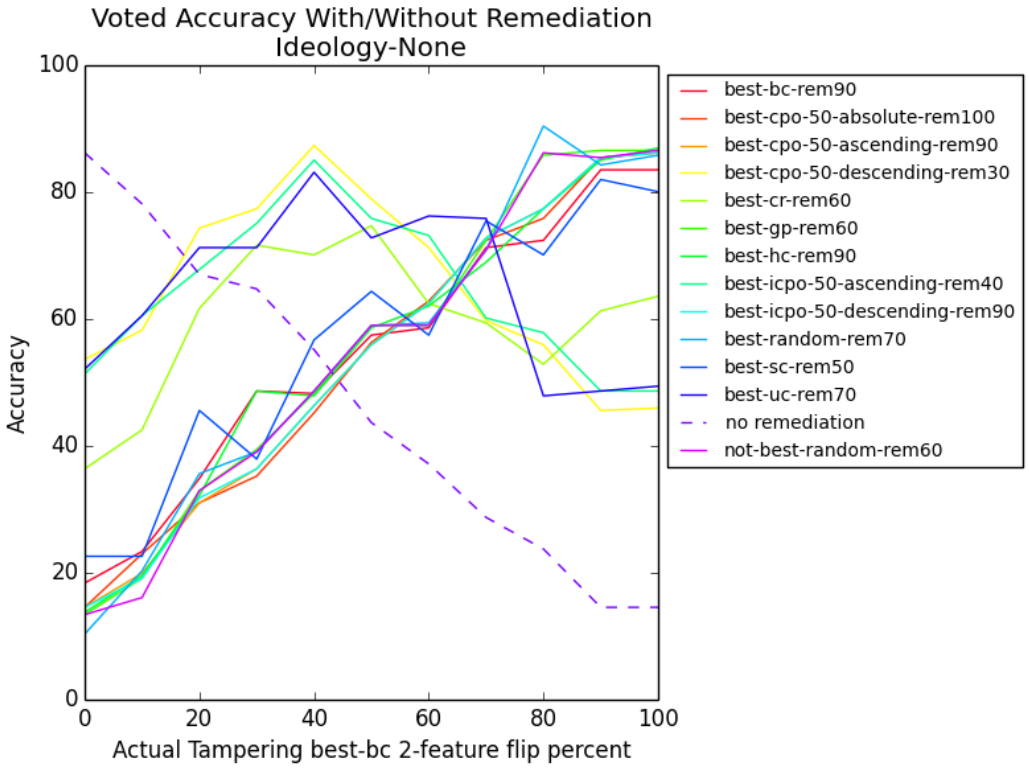


(b) Presumes Brute Clustering Tampering

Figure 15: Remediation Diagonal Accuracies

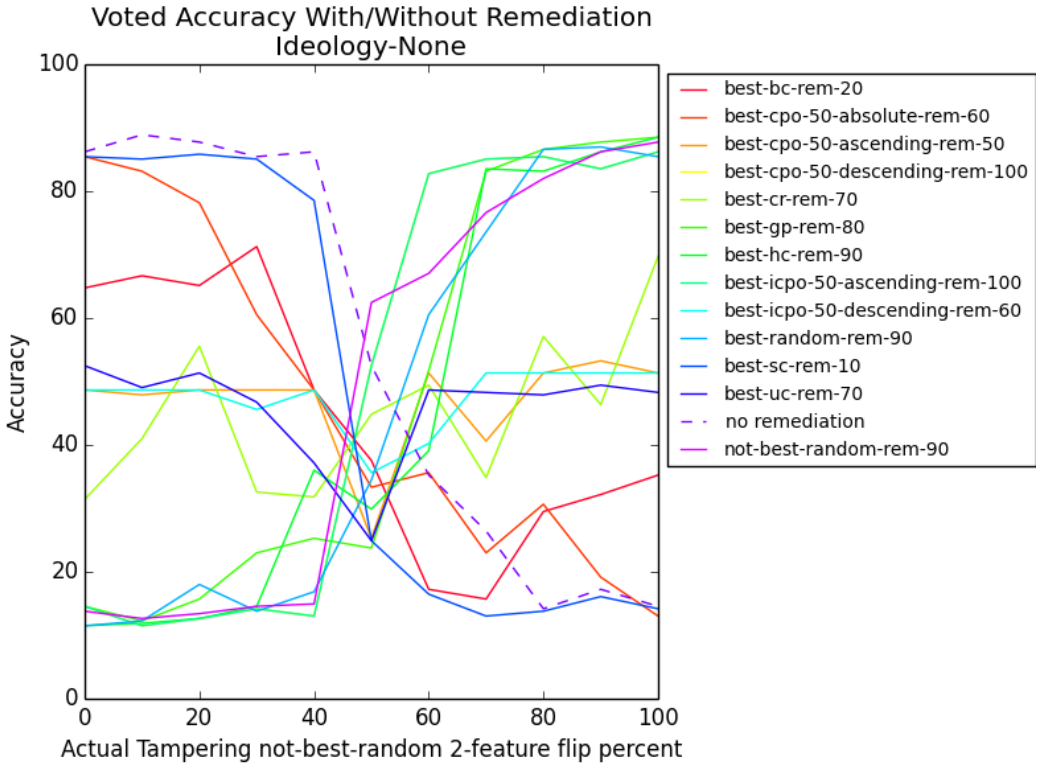


(a) Presumes Random Tampering

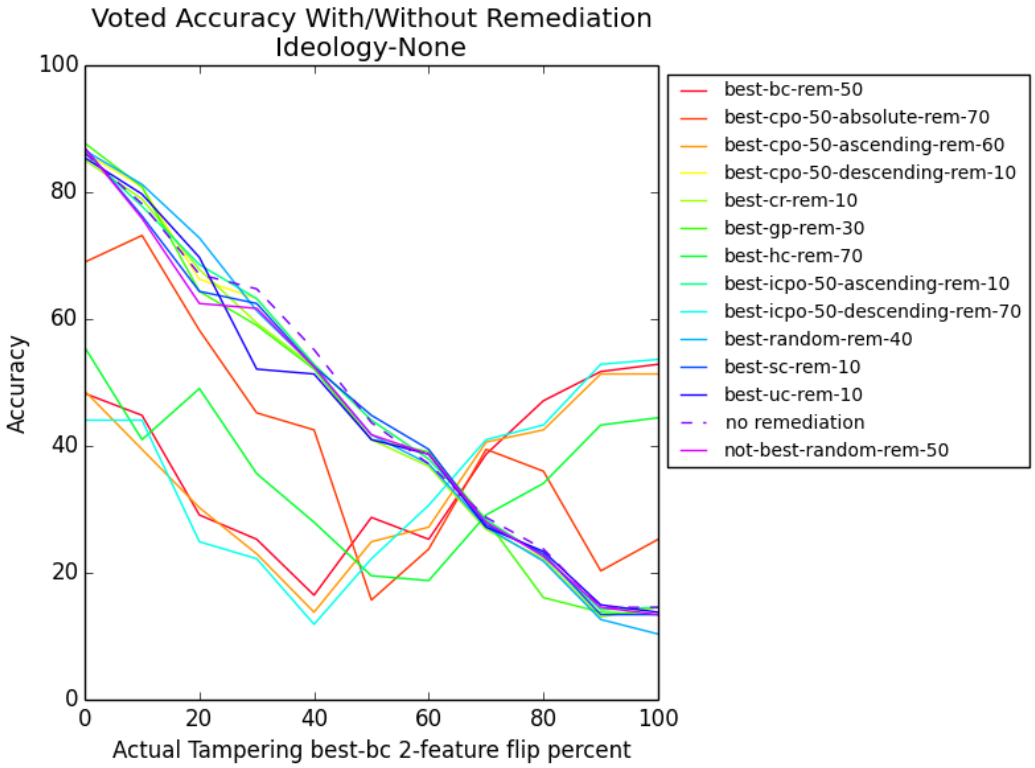


(b) Presumes Brute Clustering Tampering

Figure 16: Remediation Good Accuracies

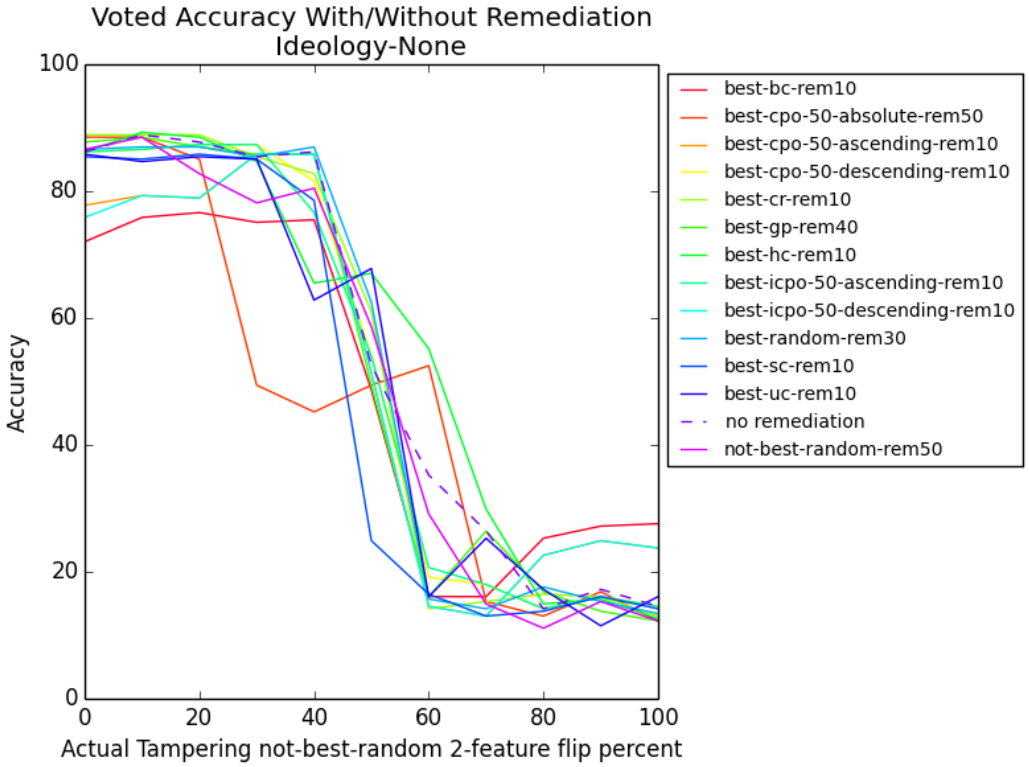


(a) Presumes Random Tampering

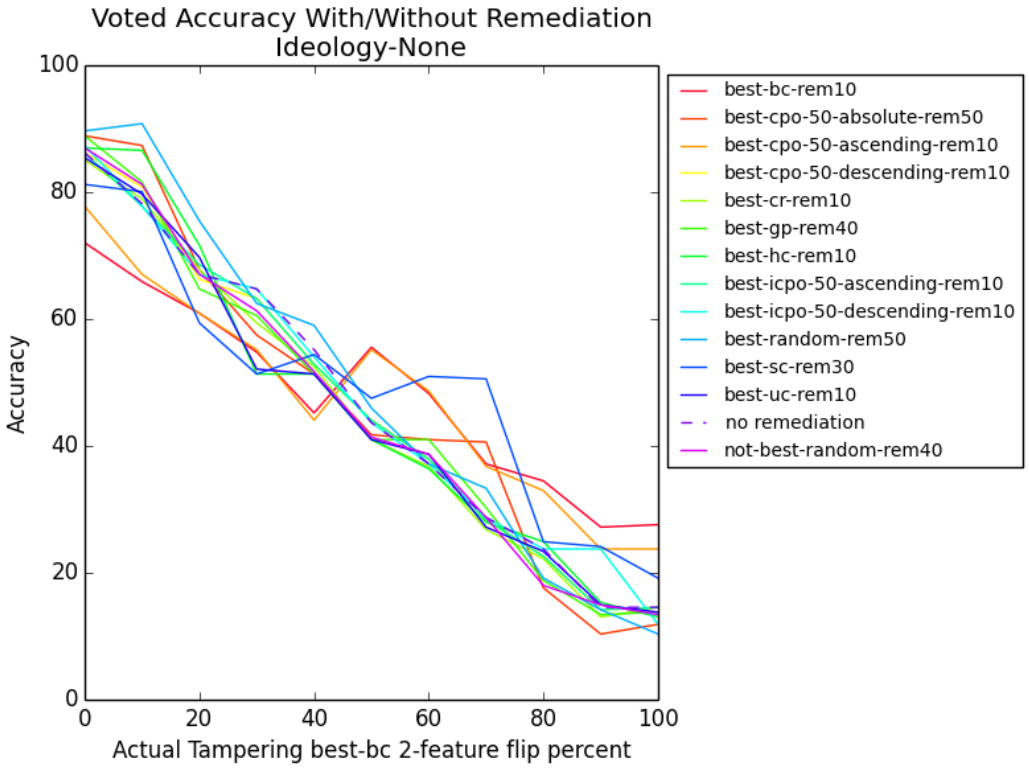


(b) Presumes Brute Clustering Tampering

Figure 17: Remediation Bad Accuracies



(a) Presumes Random Tampering



(b) Presumes Brute Clustering Tampering

Figure 18: Remediation Best 10+% Accuracies

7 Quantifying Paranoia for Label Tampering Attacks

7.1 Introduction

In the previous chapter, we used a supervised machine learning meta-model to attempt to detect *individual* points that may have been tampered with in order to undermine a lower-level supervised machine learning model.

A different but also useful and interesting question is: can we detect, *overall*, if a data set has been tampered with? Certainly if we confidently can find even one tampered point we know that tampering has happened, but it could be the case that a data set seems odd or misshapen as a whole (in a fashion we'll attempt to make quantitative below), even if we can't confidently point a suspicious finger at any one data point.

One idea, only lightly explored in CADA, was that comparing a detected tampering to an *optimal* tampering might provide a principled way of detecting a sentient signature. In other words, even if mislabeled points are detected, it might be that simple noise or other non-adversarial processes are the cause, as was indeed the case in some of the work that inspired CADA. "Never ascribe to evil what can be explained by incompetence"; to make that principle quantitative we need a statistical test that can distinguish between random attacks and adversarial ones.

As one approach to distinguish noise from adversarial attack, we note that the Ensemble of Outlier Methods in the previous chapters gives us a rank ordering of the possible actual tampering in a data set. Further, the various attacks of Section 4 tell us the order in which a *competent* attacker would attack. So if a detected tampering seems statistically more similar to an optimal attack than a random one, then this seems to suggest sentient involvement.

This seemed a promising idea, but was not fully explored. Partly because it seems to require a daunting understanding of the characteristics of the noise in the Changed/Unchanged assessments delivered by the EOM method, but also because an alternate line of attack proved more quickly successfully. Instead of attempting to analyze a noisy estimate of the purported attack, we will use Bayesian logistic regression (yet another machine learning method) to model the data, including its labels. We will then analyze the "goodness of fit" of the logistic regression model to detect, in a quantitative way, models that are indeed misshapen, models that are more ill-fitting than the noise processes in untampered data would predict.

7.2 Background

In its simplest formulation, logistic regression can be used to analyze relationships between a dichotomous (categorical) dependent variable, also called the "response" variable, and a set of independent variables. The dependent variable can take on two values e.g. {Ideology, None}, {good, bad}, or in general, $Y \in \{-1, 0\}$, while the independent variables X may be metric or categorical. The goal is to estimate the probability that a particular set of values for the independent

variables is a member of a response category, e.g. $Y = 1$ or $Y = -1$. The model here will be represented by a series of weights β, applied to the independent variables and fed through an estimation function. These weights capture the probabilistic odds of one feature being more or less significant, relative to the other features, in predicting the category of the response variable.

Specifically we are interested in the conditional probabilities:

$$P(y = 1|\beta, \mathbf{x}_i) = \psi(\beta^T \mathbf{x}_i) = \psi\left(\sum_j \beta_j x_{i,j}\right)$$

where $P(y = 1|\beta, \mathbf{x}_i)$ is the probability that observation x_i belongs to the category "1", modeled as a transformation $\psi()$ of the inner product of the weights β and the independent variables describing x_i. There are a number of alternative transformations available, but we will focus on the logistic function:

$$\psi(z) = \frac{\exp(z)}{1 + \exp(z)}$$

Logistic regression, or log-regression, combines the independent variables to estimate the probability that a particular event will occur (Figure 19). The response variable is a probability value in the interval $[0, 1]$.

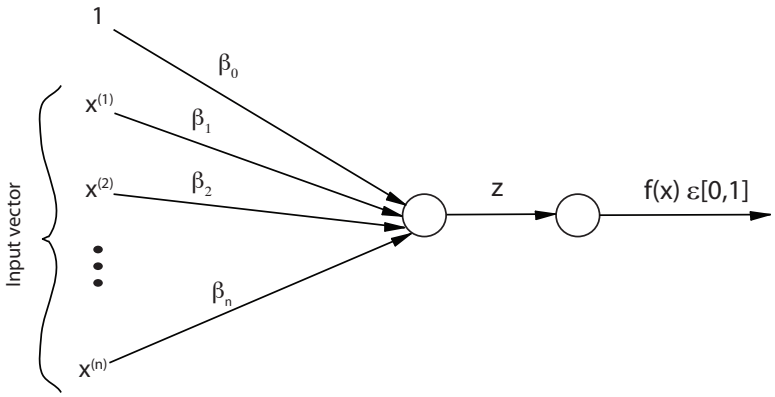


Figure 19: Data Flow in Log-regression

The only significant distributional assumption with this approach (other than assuming that the underlying data from classes 1 and -1 do indeed come from *some* parametric distribution) is that the log-likelihood ratio of the true class distribution, whatever it is, is linear in the observations. This assumption is valid if the data comes from a large range of exponential density families, e.g. normal, beta, gamma, etc. The method does not make any assumptions of normality, linearity, or homogeneity of variance for the independent variables.

If f_i are the class conditional parametric densities for classes 1 and -1, and β are the model parameters, then:

$$\log \frac{f_1(\mathbf{x})}{f_{-1}(\mathbf{x})} = \beta_0 + \beta_1 \mathbf{x} + \cdots \tag{1}$$

Equivalently:

$$f(\mathbf{x}) = \psi(\beta_0 + \beta_1 \mathbf{x} + \cdots) \tag{2}$$

where $\psi(z)$ is the logistic function $\psi(z) = 1/(1+\exp^{-z})$ and $f(\mathbf{x}) = P(y=1|\mathbf{x})$ is the probability of being in class 1 (Figure 19). The likelihood function for the data is

$$L(D,\beta) = P(D|\beta) = \prod_{j=1}^{n} P(y = y_j|\mathbf{x}_j,\beta)$$

where $\mathbf{x}_j$ is the vector of independent variables associated with the j-th observation x_j, and y_j is the label, the response value, associated with that observation.

Let $\pi_j = p(y_j = 1|\mathbf{x}_j,\beta)$. Since, remember, $y_i \in [0,1]$, the likelihood function can then be expressed:

$$L(D,\beta) = \prod_{j=1}^{n} P(y = y_j|\mathbf{x}_j,\beta) = \prod_{j=1}^{n} \pi_j^{y_j}(1-\pi_j)^{1-y_j}$$

The log-likelihood is often more convenient: $l(D,\beta) = \log L(D,\beta)$. Our goal is to maximize the likelihood function. Or, equivalently, minimize the error function: $E(D,\beta) = -\sum_i l(D_j,\beta)$.

7.3 Model

Given N observations, then for $i = 1,\ldots,N$:

$$\begin{aligned} y_i &\sim \pi^x(1-\pi)^{1-x} \\ \nu_i &= \beta_0 + \beta_1 \mathbf{x} + \cdots \\ \psi(\pi_i) &= \nu_i \end{aligned}$$

The log-likelihood function is then:

$$l_i = y_i * \log(\psi^{-1}(\nu_i)) + (1-y_i) * \log(1-\psi^{-1}(\nu_i)) \tag{3}$$

7.4 Establishing the Priors

Determining the logistic parameters requires starting with priors, with a starting point for their structure. There are three broad possibilities:

1. Parameter priors which involve L_1 regularization using a Laplace (double exponential):

$$\begin{aligned}\beta_j \sim &\quad \lambda_\beta \exp(-\lambda_\beta x)\\ \lambda_\beta \sim &\quad U(0.001, 10)\\ \beta_0 \sim &\quad N(0, 1000)\end{aligned}$$

2. An alternate structure which again involves L_1 regularization using a Laplace (double exponential):

$$\begin{aligned}\beta_j \sim &\quad \lambda_\beta \exp(-\lambda_\beta x)\\ \lambda_\beta = &\quad 1\\ \beta_0 \sim &\quad N(0, 1000)\end{aligned}$$

3. Parameter priors which are assumed to be vague or dispersed:

$$\begin{aligned}\beta_j \sim &\quad N(0, 1000)\\ \beta_0 \sim &\quad N(0, 1000)\end{aligned}$$

Case 2 provided the best convergence rates and was the basis for all CADA analyses. Further investigation into using a more general prior from the Cauchy family [**?**] could be useful.

7.5 Identifying Influential Observations

Conditional Predictive Ordinate

The use of cross-validation (CV) predictive densities is a common approach for model checking. To begin, the full data, $\mathbf{y}$, is randomly divided into two subsets $\{\mathbf{y}_1, \mathbf{y}_2\}$. The first set $\mathbf{y}_1$ is used to fit the model, and the second set $\mathbf{y}_2$ is used for model validation by checking the CV predictive density:

$$f(y_2|y_1) = \int f(y_2|\beta) f(\beta|y_1) d\beta$$

Difficulties arise since selection of different $\mathbf{y}_1$ and $\mathbf{y}_2$ generally provide different results. As an alternative, the leave-one-out cross-validation predictive density, $CV-1$, has been proposed [**?**]. This is also known as the Conditional Predictive Ordinate (CPO) [**?**]). In either case, instead of breaking the data into a single partition of $\{\mathbf{y}_1, \mathbf{y}_2\}$, the data is instead broken into y_i (the single sample) and $y_{\neg i}$ (all samples except y_i), and the CPO model checking is applied to that split for all i.

The idea is that CPO_i provides a quantitative measure of the effect of each observation y_i on the predictive density $f(y)$.

$$\begin{aligned}[f(y_i|y_{\neg i})]^{-1} &= \frac{f(y_{\neg i})}{f(y)} = \int \frac{f(y_{\neg i}|\beta)f(\beta)}{f(y)} d\beta \\ &= \int \frac{1}{f(y_i|\beta)} f(\beta)f(y)d\beta \\ &= E_{\beta|y}\frac{1}{f(y_i|\beta)}\end{aligned}$$

The CPO for each sample can be estimated from the sample mean of the inverse density function evaluated at y_i for each set of parameters β from the full posterior distribution:

$$CPO_i = \frac{f(y)}{f(y_{\neg i})} = \left(\frac{1}{N}\sum_{j=1}^{N}\frac{1}{f(y_i|\beta_j)}\right)^{-1}$$

Specifically, the CPO_i is the inverse of the posterior mean of the inverse likelihood of y_i . A Monte Carlo estimate of the CPO is obtained without actually omitting y_i from the estimation, and is provided by the harmonic mean of the likelihood for y_i .

The CPO is a convenient posterior predictive check because it may be used to identify outliers and influential observations. The CPO expresses the posterior probability of observing the value of y_i when the model is fitted to all data except y_i, with a higher value implying a better fit of the model to y_i, and very low CPO_i suggest that y_i is an influential observation. *Influential* in this case indicates that the points are outliers. Removing (or changing the labels on) these points will change the estimates provided by the model. This, then, is the source of the "cpo" and "icpo" attacks discussed in Section 4.3.

In our case, since the results are discrete, CPO_i is an estimate of the probability of observing y_i given that $y_{\neg i}$ has already been observed. Therefore comparison of CPO_i with the corresponding frequencies of $y_{\neg i}$ provides insight into the predictive capability of the model. Closely related is the posterior predictive ordinate for the i^{th} observation, $PPO_i = f(y_i|\mathbf{y})$.

PPO_i also provides a measure of the influence of an observation. Low PPO_i values are generally associated with the tail area of the sampling distribution, but extremely low values indicate outlier observations. A *estimate* of the conditional predictive ordinate is provided by the harmonic mean of the PPO_i:

$$\widehat{CPO_i} = \left(\frac{1}{T}\sum_{j=1}^{T}\frac{1}{f(y_i|\beta_j)}\right)^{-1} \tag{4}$$

However this can be cumbersome to calculate and so we will take advantage of the following:

$$\begin{aligned}ICPO &= [CPO_i]^{-1} \\ E[ICPO] &= \frac{1}{T}\sum_{j=1}^{T}\left(\frac{1}{PPO_j}\right)\end{aligned}$$

where PPO_i is readily available from within our MCMC simulation and the CPO_i can be estimated from the posterior predictive ordinate over T MCMC simulations[**?**].

Pseudo-Bayes Factors

In the previous section we developed a method whereby logistic regression and Conditional Predictive Ordinates were used to evaluate the degree of influence of individual observations on the overall model; being influential in this fashion suggests that a point is an outlier.

This in turn suggests that we might be able to determine if a data set, overall, has more outliers than one would expect to occur "naturally". In other words, given an untampered data set U we could start by extracting a logistic regression model from it. Call this Model A, M_a, and recall that such a model is captured in the parameter vector $\beta_{\mathbf{a}}$. Further, consider a new data set D, drawn from the same distribution, and then *possibly* subject to label tampering. Extract Model B, M_b, from it, represented by $\beta_{\mathbf{b}}$.

If D is untampered data drawn from the same distribution as U, then Models A and B should both individually have roughly the same goodness-of-fit for D, and we can check this by examining the CPO values generated by Models A and B on D.

To set this up, we start by computing the posterior model odds for Models A and B. Formally, given data D, to compare Model A: M_a with parameter vector β_a to Model B: M_b, β_b, we compute the posterior model odds and manipulate the result to separate something called the pseudo-Bayes Factor (PBF):

$$PBF_{ab} = \frac{f(D|M_a)}{f(D|M_b)} = \frac{\int f(D|\beta_a, M_a) f(\beta_a|M_a) d\beta_a}{\int f(D|\beta_b, M_b) f(\beta_b|M_b) d\beta_b}$$

The Bayes factor provides insight into the relationship between the posterior model odds and the prior model odds, that is, the model odds before the data is available:

$$\frac{f(D|M_a)}{f(D|M_b)} = PBF_{ab} \frac{f(M_a)}{f(M_b)}$$

This implies that whatever the prior odds of Model A relative to Model B, the data, through the PBF, provides additional information that will positively or negatively influence the odds of Model A versus Model B.

The PBF is readily available from our CPO analysis:

$$PBF_{ab} = \frac{\prod_N CPO_{ai}|M_a}{\prod_N CPO_{bi}|M_b} \tag{5}$$

The intuition is that if Models A and B derived from data D are indeed nearly identical, the PBF will be very close to 1, and so the data D doesn't particularly favor one model over the other.

However, if the data *has* been tampered with, then Model B will fit better, have fewer outliers. And since outliers generate lower CPO values, the PBF ratio will thus be less than one.

In sum, given our set up, a $PBF_{ab} < 1$, or, equivalently, $\log(PBF_{ab}) < 0$, suggests that the data D has been tampered with. Prior work[**?**] provides a table for how to interpret these values; remember, "support for M_b" means "the data has been tampered with".

Interpretation	B_{ab}	$\log(B_{ab})$	$p(M_a \mid D)$
Very strong support for M_b	<0.0067	<−5	<0.01
Strong support for M_b	0.0067 to 0.05	−5 to −3	0.01 to 0.05
Positive support for M_b	0.05 to 0.33	−3 to −1	0.05 to 0.25
Weak support for M_b	0.33 to 1	−1 to 0	0.25 to 0.50
No support for either model	1	0	0.50
Weak support for M_a	1 to 3	0 to 1	0.50 to 0.75
Positive support for M_a	3 to 20	1 to 3	0.75 to 0.95
Strong support for M_a	20 to 150	3 to 5	0.95 to 0.99
Very strong support for M_a	>150	>5	>0.99

Figure 20: Interpretation Scheme for Bayes Factor [**?**]

Wasserstein's Metric (Mallows Distance): An Alternative to PBF

In Section 7.6 we will present some experimental results for detecting tampering via CPO and PBF, but first we would like to note some issues with, and an alternative to, CPO when the metric is used for more formal model criticism (see [**?**] or [**?**] p.90, for discussion and alternatives):

- The harmonic mean can become numerically unstable value of x_i with very small likelihood, and subsequent large impact on CPO.
- The applicability of the central limit theorem may then also be questioned. However, since we are looking for CPO to be an indicator rather than a hard statistic, this should not be a problem.
- CV-1 avoids double usage of the data, but is difficult to estimate. Alternatively, we use the full posterior predictive distribution as an approximation if we have a substantial number of observations [**?**], p. 205-206.

Fundamentally, the pseudo-Bayes Factor in Equation 7.5 is a comparison of the probability distribution of CPO values for each model M_i, and this points to an alternative formulation. Consider the histograms of the estimated CPO values depicted in Figure 21. The three histograms depict the CPO values from a) an untampered Ideology data set, b) a mildly tampered one, and c) an aggressively tampered one. Alternatives to PBF include pseudo Bayes factors (PsBF) based on a

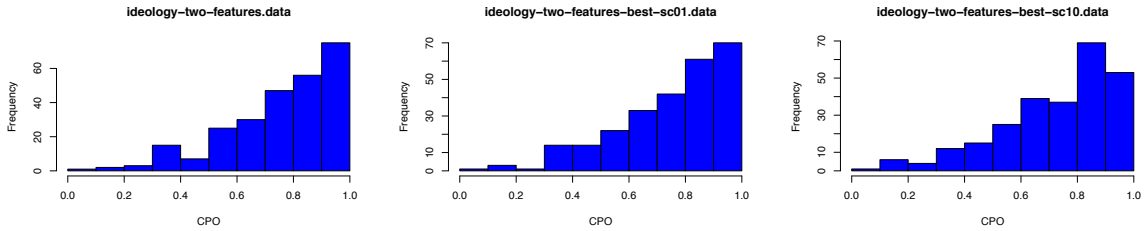


Figure 21: CPO Histograms for Model M_a, M_b and M_c

ratio of pseudo marginal likelihoods (PsMLs), Deviance Information Criterion (DIC), and Widely Applicable Information Criterion (WAIC), the latter being a recent addition to the analysis toolkit.

However, an additional alternative metric for comparison of two probability density functions is Wasserstein's Distance, also known as Mallow's Distance. Unlike the pseudo-Bayes Factor, Mallows Distance satisfies the three requirements for a true metric [**?**].

Let F, G be two distributions we wish to compare. For $F \sim (\mu_F, \sigma_F)$ and $G \sim (\mu_G, \sigma_G)$:

$$M_q(F,G) = \inf_P \left\{ (E_P||X-Y||^q)^{1/q} : (X,Y) \sim P, X \sim F, Y \sim G \right\}, \text{for } q \in [1,\infty)$$

and

$$M_2^2(F,G) = (\mu_F - \mu_G)^2 + (\sigma_F - \sigma_G)^2 + 2\sigma_F\sigma_G(1-\rho_{QQ}(F,G)),$$

where ρ_{QQ} is the QQ correlation described in [**?**] .

If F, G are multivariate non-Gaussians then: $M_2(F,G) = ||\mu_F - \mu_G|| + M_2^2(F_0, G_0)$, where F_0, G_0 are the zero mean centered F, G distribution respectively [**?**]. The first term captures the change in the location of the two distributions, while the second term captures the change in the shape from one distribution to the other.

Unlike the pseudo-Bayes Factor, Mallows Distance is expected to be less sensitive to small values of x_i that lead to a small likelihood and instability of the harmonic mean.

7.6 Experiments and Results

Feature Influence Figure 22 depicts the LogOdds for the parameters of the model described in Section 7.3 using 35 features of the 50 features for the Ideology data set. A representation of the impact of each *parameter* is present in the odds values: $LogOdds = \exp(\beta_i)$. Log-odds characterizes the percentage increase/decrease in the probability that $Y = 1$ ('ideology') given a unit increase in that feature (and holding the remaining features fixed). Features with log-odds close to zero are not having a significant impact on the model, while features with high/low log-odds are important.

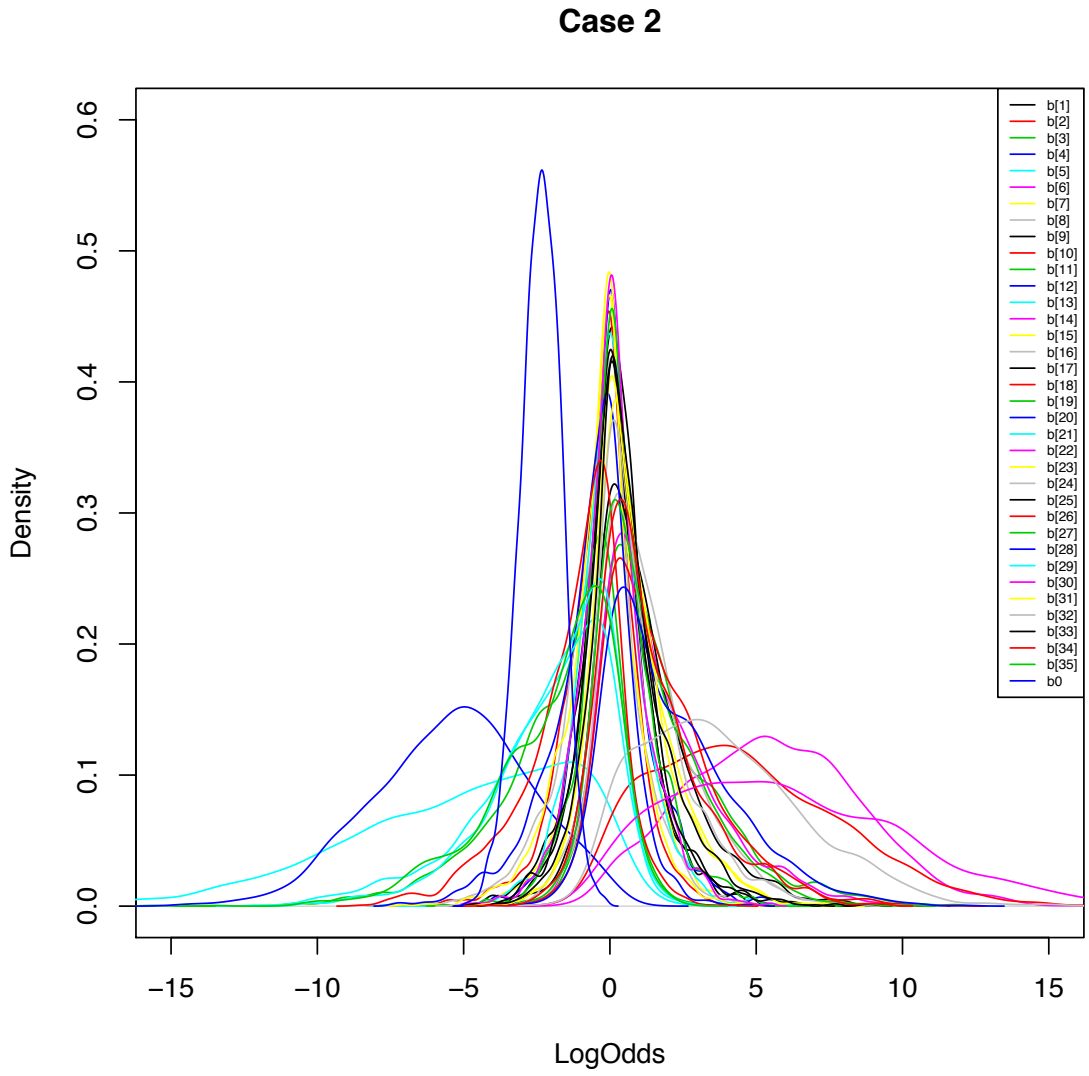


Figure 22: LogOdds Distribution for Model Parameters (35 features)

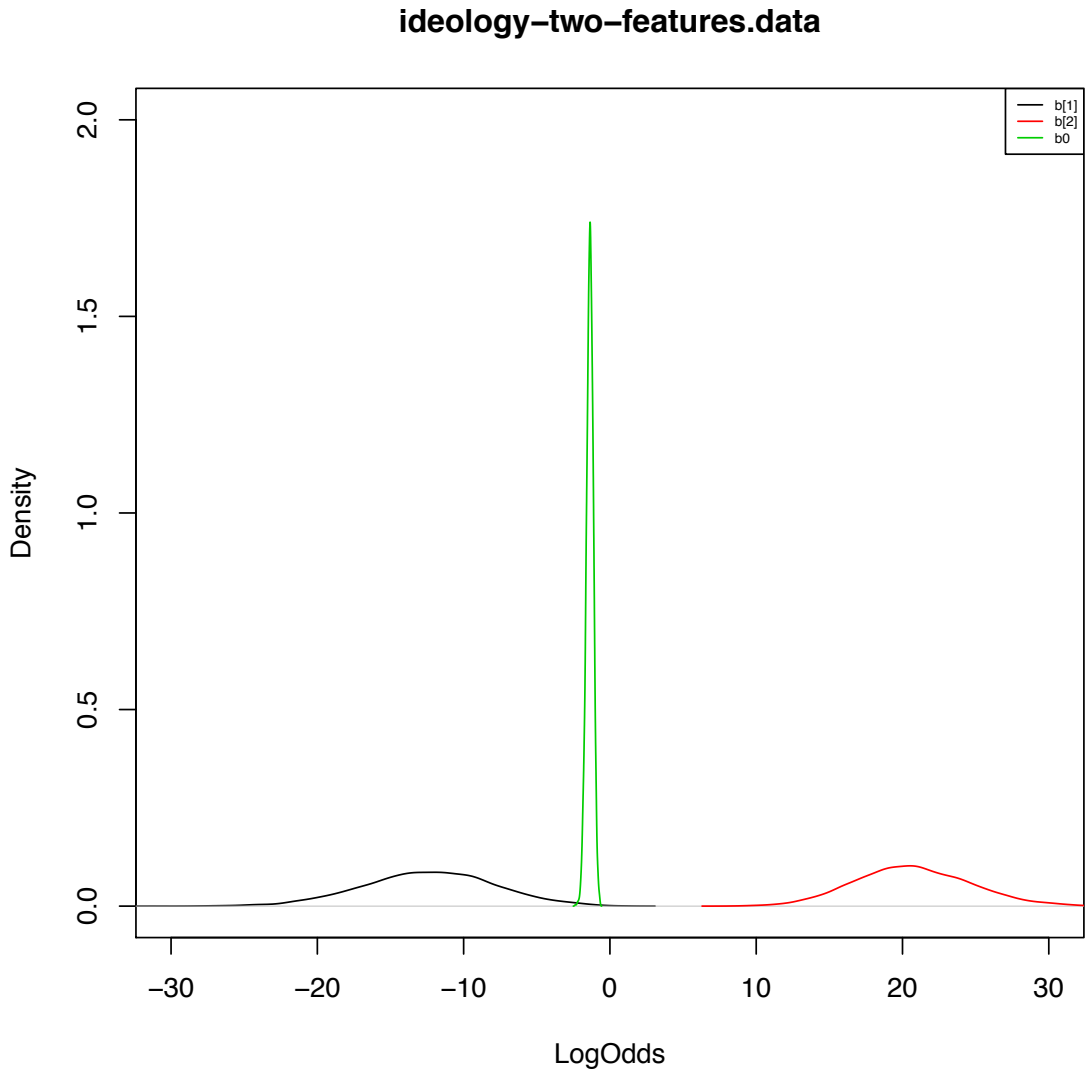


Figure 23: LogOdds Distribution for Model Parameters (2 features)

Because Figure 22 is a bit complex, consider also Figure 23, which depicts the LogOdds for the parameters of the model using only 2 of the 50 available features. To interpret this, note that it is the *position* of a peak, not it's magnitude, that matters. So:

- The green peak is near 0, which means that the choice of β_0 in the model from Section 7.3 doesn't matter much.
- The black curve, corresponding to β_1, has a peak at -11 or so, which means that feature 1 does matter, and when it is high it suggests against the Ideology label in favor of the None label.
- The red curve, corresponding to β_2, has a peak at 20. $|20| > |-11|$, so this suggests that feature 2 is more influential than feature 1, and that increases in its value vary the Ideology label.

Observation Influence Figure 24 depicts the difference between the CPO values for all 260 samples in two versions of the ideology data set, in both cases described by only two features. The difference between these two data sets is that the first data set has only one tampered label, where the second has eleven additional tampered labels. The effect of tampering, the increase in "outlierness" in the samples, is obvious, as the eleven observations where labels have been flipped are clear from the figure.

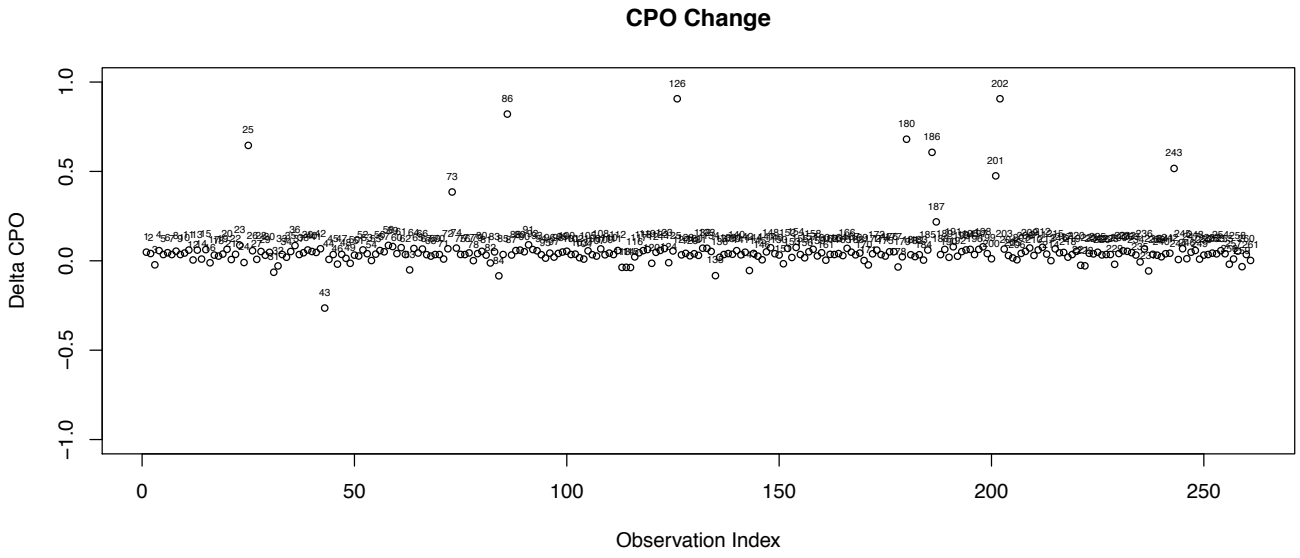


Figure 24: Identification of Tampered Observations (typical)

Tampering Indicators To determine the experimental ability of these tests for overall tampering detection, we built a model, M_A, on the untampered version of the "ideology.data" data set. We

then extracted alternate models, M_b, from a variety of versions of the "ideology.test" data drawn from the same distribution as "ideology.data". (For all experiments we used the version of the data sets that used only the best two features.) The variants were:

Untampered: No change to "ideology.test". We hope that here the PBF tests will indicate no tampering.

Random: The random attack, with budgets of 1 through 12. The point of multiple budgets is to be able to note when the attack first becomes evident. Random is a fairly incompetent attack, and so we might be slow to detect the tampering.

CPO: The Conditional Predictive Ordinate Attack, budgets of 1 through 12. This is a fairly competent attack, and so we expected to be able to detect it quickly.

SC: The Subtle Clustering attack, budgets of 1 through 12. This is nearly as effective as CPO, but it is also not easily spotted via cross-validation assessment on the training data, as discussed in Section 4.5. We were interested in whether this would increase the difficulty of spotting it via PBF.

Table 12: PBF Comparison of Three Attacks on Ideology

Budget	Random	CPO	SC
0	-0.19		
1	-3.55	-4.17	-2.17
2	-5.11	-8.57	-4.77
3	-6.96	-12.21	-6.07
4	-11.44	-15.68	-5.99
5	-15.62	-18.03	-8.06
6	-17.43	-19.80	-10.13
7	-20.67	-20.77	-11.72
8	-23.00	-22.60	-13.70
9	-24.64	-24.12	-13.64
10	-24.26	-26.82	-13.33
11	-24.93	-28.10	-14.96
12	-26.65	-29.70	-16.57

Table 12 contains the results. To interpret the values, refer back to the $\log(B_{ab})$ column of Figure 20. Some notes on interpretation:

- When there is no tampering, "Budget=0", the PBF is -0.19, which close to 0 and anyhow greater than -1, indicating only very weak support for tampering.
- With only a single data point tampered, both Random and CPO generate positive support for tampering, and once two points had been tampered they both generated strong support for tampering.

- Further, at almost all budget levels, the absolute value of PBF for CPO was higher than for Random, confirming our expectation that CPO is a more effective, a more noticeable, attack than Random.
- With a budget of 1, SC generates only weak support, and with a budget of 2 it generates positive support. SC requires a budget of 3 to register as strong support for tampering, and in general had a much smaller absolute PBF than Random or CPO, indicating that lack of effect on training-set cross validation translates into lack of impact on PBF.

Overall, it was surprisingly easy to spot tampering with the PBF test; even the most subtle attack registered after only three data points were tampered with. Admittedly, with only 260 test points, three data points are slightly more than 1% of the data, but still, this seems a usefully sensitive test.

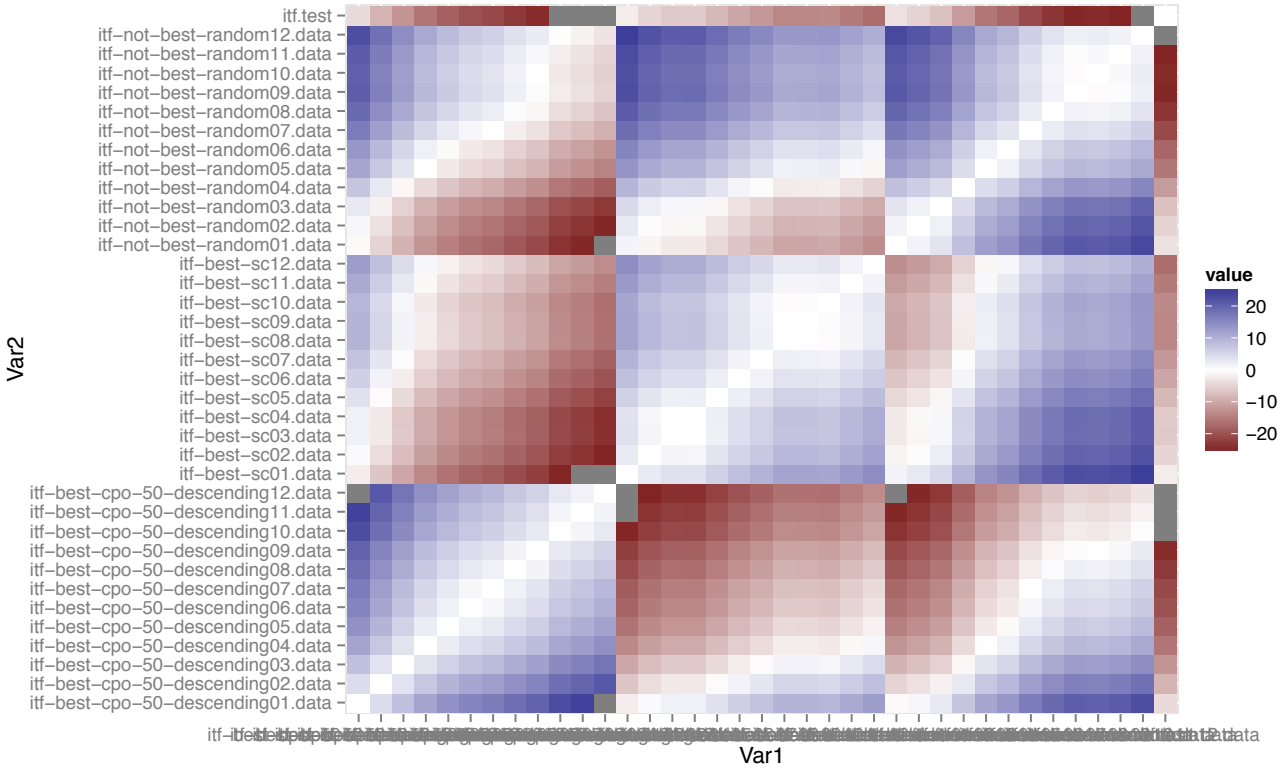
Finally, note that Table 12 compares only an untampered dataset to datasets with various levels of tampering. This is certainly the operational environment most likely to be interesting, but for a sense of the general robustness of the test, one could compare datasets corrupted by various kinds and levels of tampering. And easily so, in our case, as those results are generated as a side effect of the primary comparison.

Consider the heat map in Figure 25a, which depicts the pairwise comparison between the level of tampering on each of the corrupted datasets. The darker the color (red or blue) the higher the ratio of their conditional densities. Moving down the last column (or across the first row) provides the same results depicted in Table 12. Entries with very light color indicated tampering that is harder to detect, with white indicating that no tampering could be detected.

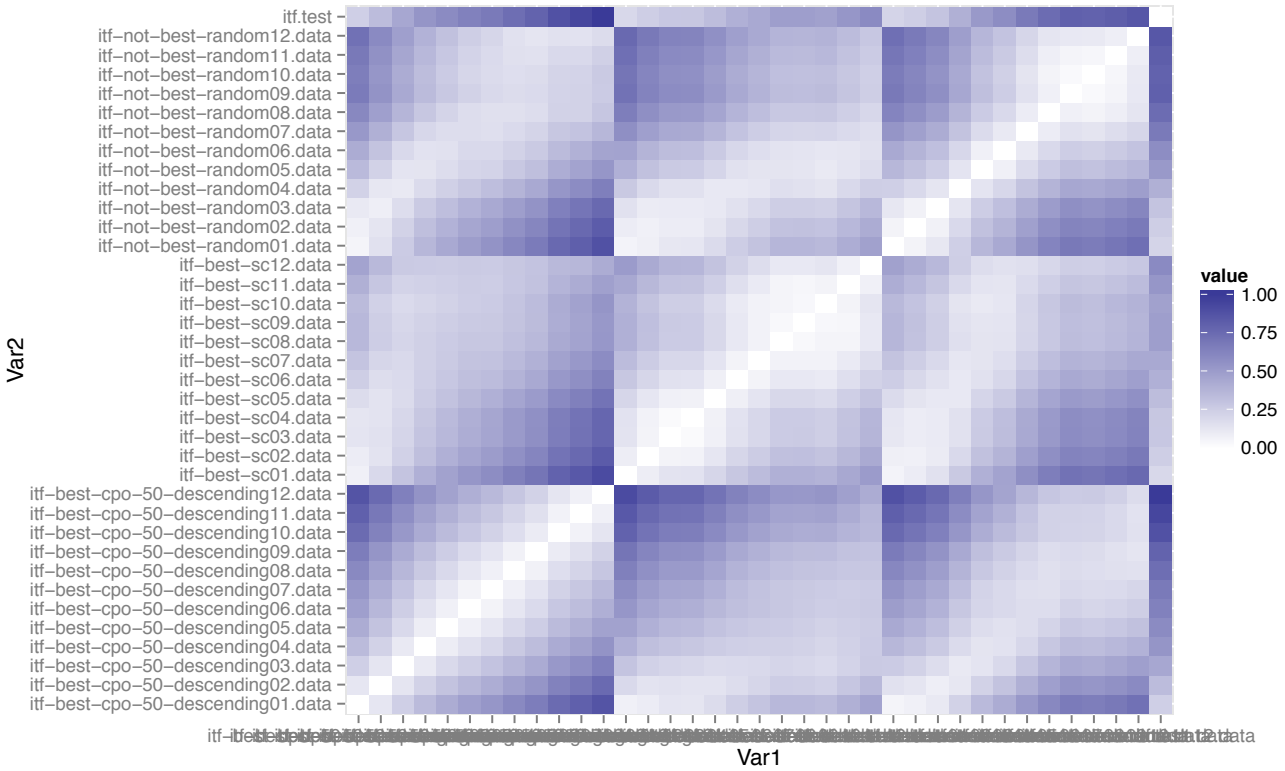
Similarly, the heat map in Figure 25b depicts the pairwise comparison between the level of tampering on each of the datasets using the Wasserstein or Mallows Distance as an indication of tampering. The Mallows Distance has been normalized to the interval $\{0, 1\}$. The darker the blue color the easier it is to detect a difference in the labels. Moving down the last column provides a summary of tampering detection relative to the test data. Entries with a very light color indicated tampering is harder to detect, with white indicating that no tampering could be detected. As can be seen, all tampering relative to the test data was detected.

Mallow's Distance provides a somewhat more sensitive measure of tampering than PBF. This sensitivity is evidenced in comparing the two heat maps. Consider the top left group of cells in Figures 25b and 25a. The broad diagonal of white subcells indicate that PBF has difficulty in identifying an increase in the level of tampering between two data sets. (e.g. itf-not-best-random09.data versus itf-not-best-random10.data). Examining the same area on Figure 25b indicates that the metric based on Mallows distance does however capture the increase in tampering.

The grey cells in Figure 25a, indicate comparisons where there are numerical difficulties in calculating the pseudo-Bayes factor; it becomes overwhelmed by the differences between the models. Alternatively, as depicted in Figure 25b, the Mallows metric does not experience these issues.



(a) Pseudo-Bayes Factor Heatmap



(b) Scaled Mallows Distance Heatmap

Figure 25: Difference Detection: All vs All

8 Cluster Tampering Via Data Mines

8.1 Introduction

Supervised vs Unsupervised Methods in CADA

In prior chapters we have primarily focused on supervised machine learning, partly because it permits crisp accuracy metrics that allow a quantitative assessment of the effect of an adversary's tampering with one's data. Clustering is an important and widely used form of *unsupervised* machine learning, and so we investigated it as well in the CADA project, for completeness' sake. Since supervised and unsupervised methods are so different, we expected the CADA investigations thereof to be fairly distinct as well. We were surprised, then, to find a substantive amount of commonality in ideas and algorithms; we shall even end this section with a demonstration that remediation via ensembles of outlier methods is applicable and useful even here.

Clustering, Plagiarism, Attacks, Defense: A Summary

Clustering is a useful tool for analyzing unlabeled data. It can be used to find plagiarized Android apps [**?**], to classify network traffic [**?**], to identify similar queries at search engines [**?**], and for many other applications. Rather than cluster "controlled data", clustering is often applied to "found data". In the former case, the data is collected from physical measurements such as gene expression for gene analysis [**?**]. In the latter case, the origin and integrity of the data is unclear. Depending on the purpose of the clustering, an adversary may tamper with the data in order to subvert the clustering algorithm.

In the case of plagiarized Android applications (apps), a plagiarist may seek to subvert the clustering algorithm in at least two ways. First, she can try to manipulate the apps that she copies in such a way that the copied app is no longer similar to the original app. We call this form of attack an *evasion attack*. For example, spam emails will often copy characteristics of normal emails to try to avoid detection. Second, she may seek to manipulate the cluster structure by adding specifically crafted apps (which we call *data mines*). This form of attack may also allow a plagiarizer to evade detection, however, we explore how it may be used to poison the clustering to undermine confidence in the tool. For this reason, we call this form of attack a *confidence attack*.

In this section, we explore and evaluate the effectiveness of the proposed confidence attack. First, we describe how an attacker may select an ordering of clusters to merge. In a real-world scenario, we assume that an attacker would have a specific goal, however, we evaluate many orderings to develop intuition about possible attacks and evaluate which degrades the accuracy of plagiarism detection the fastest. Then, we discuss how *bridges* can be generated using data mines to arbitrarily merge clusters. These bridges span the gaps between clusters with sufficient density to lead the clustering algorithm to interpret the data as a single cluster. Next, we measure how the quality of the clustering degrades as a function of the number of data mines an attacker creates.

Finally, we propose an additional remediation phase that the defender can use to prune data mines from her dataset based on outlier measurements.

Our contributions are as follows:

- We present a methodology for selecting and then merging arbitrary clusters.
- We evaluate the effectiveness of various attacks in a real-world scenario.
- We propose metrics for attacker and defender cost and measure the trade-offs.
- We find the clustering algorithm DBSCAN alone is insufficient for adversarial settings.
- We propose a remediation methodology to remove data mines from a dataset based on outlier measurements.

8.2 Background

DBSCAN

DBSCAN [**?**] is a density-based clustering algorithm for large spatial databases. It clusters points using two parameters: T and $MinPts$. T is the distance threshold that determines the size of each point's neighborhood. $MinPts$ is the number of neighbors a point must have in its T-neighborhood in order to be considered a core point. A cluster is defined as all the points that are *density-reachable* from a core point p. In order for a point, q, to be *density-reachable* from core point p, it must either be in the T-neighborhood of p or there must be a series of core points $p_0, \ldots, p_n$ such that p_0 is in the T-neighborhood of p, p_i is in the T-neighborhood of $p_i - 1$ $(0 < i <= n)$, and q is in the T-neighborhood of p_n.

DBSCAN is an attractive clustering algorithm for a number of reasons. First, it does not require the number of clusters to be specified ahead of time. Second, combined with locality sensitive hashing (LSH) [**?**] that can approximately identify the nearest-neighbors of a point in logarithmic time, DBSCAN can cluster all points in linearithmic time.

AnDarwin

AnDarwin [**?**] is a pre-existing tool developed to detected cloned Android apps. Android apps are distributed through centralized markets, such as Google Play. The majority of apps [**?**] are available for free but include ads for monetization. Unfortunately for app developers, apps are often cloned by plagiarists who alter the app to redirect the ad revenue stream into their own accounts [**?**].

AnDarwin detects cloned apps on a large scale. It represents each app as a set of features as follows: 1) it converts Android's DEX bytecode to Java bytecode, 2) it uses WALA [**?**] to compute

program dependency graphs (PDGs) for methods found within the app, 3) it encodes the PDGs as *semantic vectors* that it clusters to form features and, finally, 4) it represents each app as a set of binary features where a feature, f, is set if an app contains a PDG whose semantic vector clusters into cluster f. An overview of this process is shown in Figure 26. AnDarwin uses MinHash [**?**, **?**] to identify the T-neighborhoods of individual apps (this is a LSH algorithm that allows AnDarwin to avoid computing a pairwise similarity matrix). Finally, to identify clusters of similar apps, it assigns each app to its own cluster and then processes the pairs of apps identified by MinHash (the individual app and each of its neighbors) and merges the clusters containing each of the pair of points. Clusters of a single app are ignored.

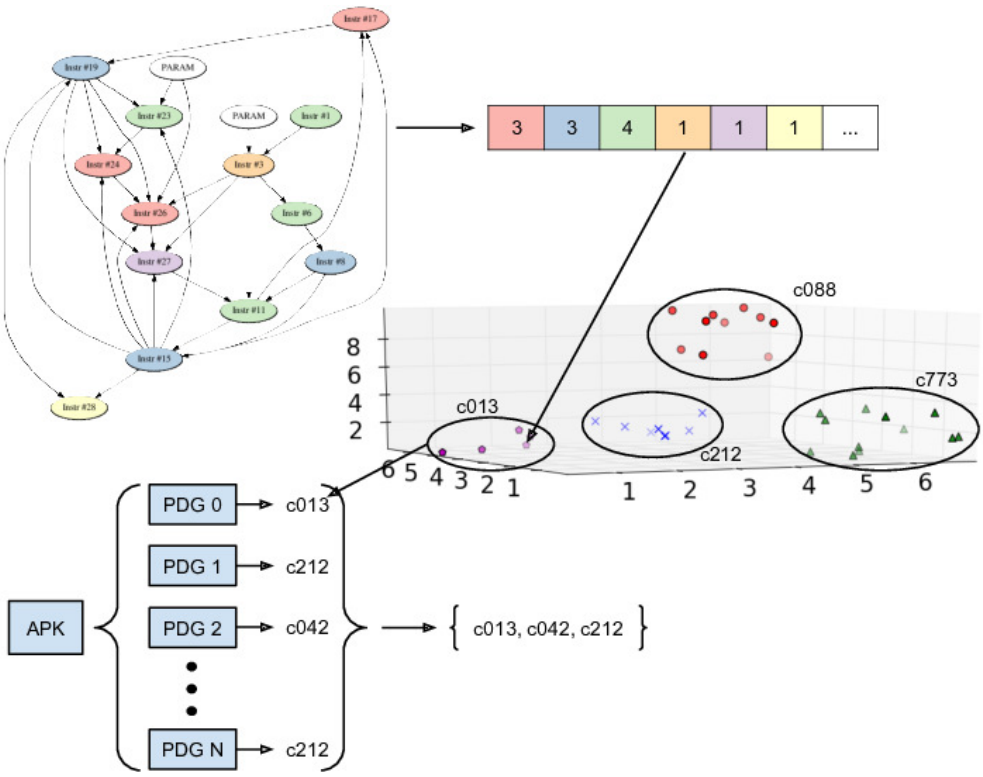


Figure 26: Overview of how AnDarwin represents Android apps as a set of features. First, it extracts program dependency graphs (PDGs) from apps, such as the one shown in the top left. It then computes a *semantic vector* for the graph which is a frequency vector for the different node types within the PDG. It then clusters the semantic vectors for all PDGs from all apps and treats each cluster of semantic vectors as a feature. Finally, it represents each app as a set of features based on which PDGs the app contains.

Although not stated in the original work, this process for building clusters is equivalent to DBSCAN with a $MinPts$ value of 2, where every point with at least one neighbor is a core point. In the original DBSCAN paper, the authors suggest using values for T and $MinPts$ that represent the "thinnest" cluster that should not be considered noise. This is problematic for AnDarwin, as AnDarwin's goal is to find app plagiarism which could consist of just one original app and one (similar) plagiarized app. If AnDarwin uses a value larger than $MinPts = 2$, then AnDarwin may miss many instances of app plagiarism.

8.3 Related Work

There have been several works that study the affects of adversarial input on machine learning. In a supervised context [**?**, **?**], the problem is often analyzed using game theory. Specifically, these works seek to find an equilibrium between the defender who uses a classifier and one or more attackers. In this work, we focus on unsupervised machine learning. Others have looked at adversarial input on unsupervised machine learning. Notably, Dutrisac et al. [**?**], outline a similar attack as the one outlined in this paper – the process of bridging the gap between two clusters to merge them. In this work, we explore this process using DBSCAN as the clustering algorithm. We also formalize the attacker cost based on how many points she must generate in order to bridge gaps. Finally, we explore how effective these attacks can be using a dataset and tool with real-world applications and whether the attacks can be remediated with outlier measurements.

Biggio et al. [**?**] present a framework for performing security evaluations of clustering algorithms. They develop an *adversarial clustering* theory that includes modeling the attackers goals, knowledge, capabilities, and strategy. They explore two forms of attacks: *cluster poisoning attacks* and *obfuscation attacks*. These are analogous to the confidence and evasion attacks described in the introduction. To demonstrate their framework, Biggio et al. evaluate single-linkage clustering. Specifically, they show how an attacker can use a small number of points, heuristically chosen, as bridges to poison the clustering. In this work, we present an equation for the number of points required to bridge two clusters for DBSCAN and perform an extensive evaluation of the attacks against AnDarwin. Additionally, we show that the attacks apply to density-based clustering algorithms.

DBSCAN's clustering algorithm is similar to single linkage clustering that is used for agglomerative clustering. A well-known issue with single linkage clustering is the chaining phenomenon. The chaining phenomenon occurs because the algorithm merges two clusters even when there is only a single pair of points that are similar between them. We exploit this weakness when building bridges to span the gap between clusters. Other linkage methods, such as complete-linkage clustering, avoid this phenomenon but may create many smaller clusters.

There have been several algorithms proposed to improve on the original DBSCAN algorithms. Notably, C-DBSCAN [**?**] adds constraints (*Must-Link* and *Cannot-Link*) to the original DBSCAN algorithm. These constraints improve on cases where DBSCAN performs poorly: if clusters are partially overlapping, connected by bridges, or have very different densities. Constraints are formed using domain knowledge although Ruiz et al. found that even random constraints can improve the clustering performance. The C-DBSCAN work provides an interesting defense against the attacks described in the current paper, however, it requires human insight to build the constraints.

8.4 Threat Model

We assume that the attacker can generate arbitrary points in feature space and that the attacker can inject those points into our dataset. For example, in the case of the Android apps, the attacker

could generate apps with arbitrary feature sets by copying methods from apps whose features the attacker wishes to include into her app. Since AnDarwin does not perform any dead code analysis, each of these injected methods is treated as a regular feature regardless of whether it is needed for app functionality or not. This exploits the semantic gap between program analysis and program execution [**?**]. In order to get her apps into our dataset, the attacker could create an account on a third-party Android market and upload her apps there for us to crawl.

In terms of Biggio et al. [**?**], the adversary has perfect knowledge. The attacker knows the complete dataset, the feature space, the algorithm, and the algorithm's parameters. We discuss the feasibility of this attacker model in Section 8.8.

Though our methodology applies to all tools based on DBSCAN, to be concrete in the current work we use AnDarwin as a specific application of DBSCAN. This allows us to more concretely discuss the generation of the data mines used to merge clusters. Specifically, we assume that points represent a set of binary features and that points are compared using their Jaccard similarity.

In this work, we explore a *confidence attack* that an attacker might use against a clustering tool in order to undermine the defender's confidence in the tool and underlying algorithms. If the defender's confidence is undermined, she may not trust the tool's results and/or abandon use of the tool completely. To provide a concrete objective for the attacker, we investigate how the attacker's injected points degrades the accuracy of plagiarism detection. Specifically, for a given clustering, we can compute whether an app is an original or plagiarizing (based on methodology from Gibler et al. [**?**]) and compare that label to the label for the same app in the untampered clustering. From these labelings, we can compute the overall accuracy of plagiarism detection in the presence of some number of cluster merges.

8.5 Methodology

In this section, we describe two critical components of how an attacker would carry out her attack: 1) the order in which she picks clusters to merge, and 2) the process of generating *data mines* that will cause DBSCAN to merge her selected clusters. Before describing the mechanisms, we first outline how we identify which apps are originals and which are plagiarizing based on a clustering and then describe the metrics we will use to measure how much the attacker's cluster merges degrade the clustering. Finally, we describe how we remove data mines from the dataset based on outlier measurements.

Identifying Plagiarism

In order to identify plagiarizing apps, we leverage the owner merging methodology from Gibler et al. [**?**]. Specifically, we seek to partition apps in a given cluster based on the owner that published the app. We determine ownership in two ways: 1) the developer account name that is associated with the app when it was crawled, and 2) the public key fingerprint for the private key that the owner used to cryptographically sign the app. If two apps share either of these two identifiers, we

consider them to be from the same owner. Once we have partitioned a cluster into apps from the same owner, we then assume that the owner with the largest number of apps is the original owner and that all others are plagiarizing. While this may not always be accurate, it does ensure that we do not overestimate the number of plagiarizing apps.

Clustering Performance

In order to quantify how much the clustering degrades as the attacker merges clusters, we compute four relative performance metrics for the clusterings. These performance metrics are all supervised; they compare the clustering after some number of merges to the original clustering. The first three metrics are generic clustering comparison metrics while the last is application-specific:

- Homogeneity: measures the number of clusters in the new clustering that have all points from the same cluster in the original clustering.
- Adjusted Rand Index: measures the difference between observed and expected values in the contingency matrix. The contingency matrix measures the intersection of clusters between two clusterings.
- Adjusted Mutual Info: also uses the contingency matrix but measures the dependence between cells of the matrix using mutual information. This relates to the probability of knowing the structure of one clustering given the other.
- Plagiarism detection accuracy: we compute a standard confusion matrix for the original or plagiarizing labels given to apps as described the previous section. We then compute the accuracy of plagiarism detection using the sum of the diagonal of the matrix divided by the sum of all cells in the matrix.

The three standard clustering comparison metrics are calculated using scikit-learn [**?**]. All four metrics have 1.0 as a perfect score, and make no assumptions about the cluster structure. Homogeneity and plagiarism detection accuracy are not normalized with respect to random labeling.

Merge Ordering Algorithms

An attacker performing a confidence attack may choose any order to merge clusters. As stated in Section 8.4, we assume that the attacker wishes to optimally degrade the accuracy of plagiarism detection. Therefore, we propose the following merge algorithms to develop intuition about how different orderings may affect the degradation of the clustering quality:

- **Random:** Cluster pairs are selected at random.

- **Nearest-neighbor:** An initial seed cluster is chosen at random and then clusters are chosen in order of decreasing similarity to the seed cluster.
- **Cluster Similarity, Decreasing and Increasing:** Cluster pairs are selected based on their similarity. For decreasing, this will start by merging related clusters before merging unrelated clusters. Note: the similarity of clusters is computed as the minimum similarity of any two points that span the clusters.
- **Cluster Size, Decreasing and Increasing:** Cluster pairs are selected based on the size of the merged cluster they would create. For decreasing, this will start by merging the two largest clusters. For increasing, this picks pairs such that the merge produces a cluster of the smallest size possible (not including points to merge the two clusters).
- **Original Size, Decreasing:** Similar to the previous algorithms except instead of using the size of the cluster, the algorithm uses the number of original apps in the cluster. Assuming that the authors do not overlap between the clusters, this will cause the most original apps to be labelled as plagiarizing at each stage of merging.
- **Author Size, Decreasing:** This algorithm starts by finding the author with the most apps across all clusters and merging clusters containing all her apps first. Then, it proceeds to merge in the remaining clusters from biggest to smallest, by cluster size.
- **Greedy Pessimal - Accuracy:** Find the two unmerged clusters that, when merged, degrade the plagiarism detection accuracy the most. Repeat this process until all clusters have been merged.

During the merge ordering algorithms, the similarity is not recomputed after each merger (even though the data mines may influence unmerged clusters' similarities). To ensure that the orderings do not contain obvious redundant merges, the algorithms keep track of which clusters have been merged and will skip pairs that have already been merged.

Due to the computational complexity, we do not try to evaluate all merge orderings. For each of the above algorithms, there may be many merge orderings that can be produced. As a trivial example, different random seeds will lead to completely different random merge orderings. For many of the algorithms, the ordering is dependent on the way ties are broken. Instead of evaluating all merge orderings, we take one ordering generated from each of the above algorithms as representative of that type of attack. We anticipate that the greedy pessimal algorithm will degrade the plagiarism detection accuracy the fastest, however, it may not produce an optimal ordering. We leave exploring whether there are particularly pernicious merge orderings to future work.

Data mine generation

To merge two clusters, the adversary must change the dataset so that two previously distinct clusters meet the criteria to be a single cluster (a core point in one cluster is density-reachable from the other). She does this by generating a series of data mines between the two clusters that, to the

DBSCAN algorithm, look like core points. Based on the DBSCAN algorithm, this will merge the original two clusters. For now, assume that the *MinPts* parameter used by DBSCAN to determine the minimum neighborhood size of a core point is 2. This effectively makes every point with at least one neighbor a core point as points are in their own T-neighborhood.

Let p_S and p_T be two points, the start and target points, in different clusters that the attacker wants to merge. In order to merge these clusters, she must generate a series of $n-1$ (n will be discussed below) data mines $(p_1, \ldots, p_{n-1})$ such that:

$$\forall i \in [1,n) : Dist(p_i, p_{i+1}) \leq T \tag{6}$$
$$Dist(p_S, p_1) \leq T \tag{7}$$
$$Dist(p_{n-1}, p_T) \leq T \tag{8}$$

For notational convenience, and without loss of generality, let $p_0 = p_S$ and $p_n = p_T$. Then Equation 6 can be rendered more compactly as:

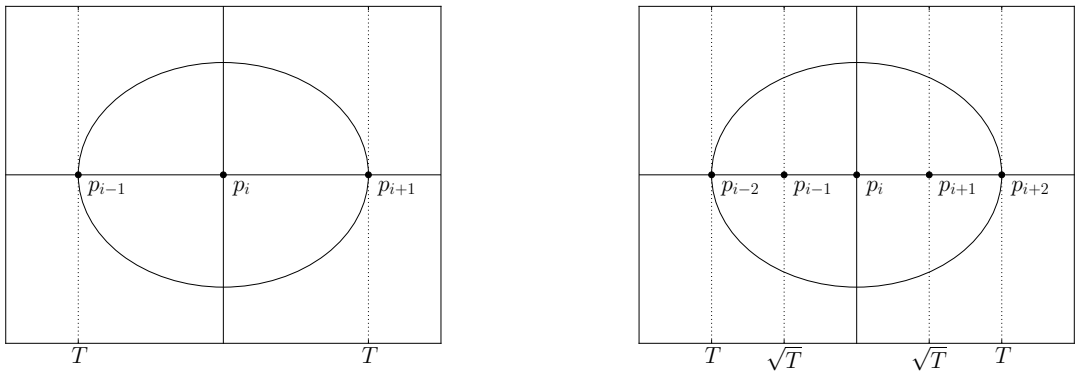
$$\forall i \in [0,n) : Dist(p_i, p_{i+1}) \leq T, \tag{9}$$

where *Dist* is an arbitrary distance function and T is the threshold used for DBSCAN to determine the neighborhood size. With a *MinPts* value of 2, this series of mines merges the two clusters that p_0 and p_n were in, achieving the attacker's goal.

Clearly, the number of data mines $(n-1)$ an attacker must craft is proportional to T: the smaller the value of T, the more mines the attacker must generate. If n were sufficiently large, say in the thousands, we may discount this as a too noisy for real-world use. To determine if this is the case, we analyze how the choice of T affects n. To minimize the number of data mines, an adversary should create points such that:

$$\forall i \in [0,n) : Dist(p_i, p_{i+1}) = T \tag{10}$$

Figure 27a depicts the geometry of these relationships. In Section 8.4, we made the assumption that instead of using the distance function, *Dist*, we are using a similarity function, Jaccard Similarity (J), and that each point, p_i, is represented by a set of features. Instead of letting T representing a maximum distance for determining the neighborhood size, let it represent a minimum similarity for the same purpose. Further, we make the worst case (for the attacker) assumption that the two points to be merged have completely disjoint feature sets. Then, the attacker can generate each mine, p_i, using a portion (x) of the features from p_{i-1} and adding a portion $(1-x)$ of the features from p_n:



(a) Three-point Chain

(b) Five-point Chain

Figure 27: Geometry for data mines. The three-point chain shows how to generate data mines when $MinPts \leq 3$ and the five-point chain shows how to generate data mines when $MinPts = 5$.

$$p_i = xp_{i-1} + (1-x)p_n \tag{11}$$

Assuming that the adversary knows T, and that p_0 and p_n are approximately the same size (which means that $|p_{i-1}| = |p_n|$ for all i), x will depend how dissimilar the intermediate data mines can be, which is a function of T:

$$J(p_{i-1}, p_i) = \frac{p_{i-1} \cap (xp_{i-1} + (1-x)p_n)}{p_{i-1} \cup (xp_{i-1} + (1-x)p_n)} \tag{12}$$

$$= \frac{|xp_{i-1}|}{|p_{i-1}| + |(1-x)p_n|} \tag{13}$$

$$= \frac{x}{2-x} = T \tag{14}$$

$$\Rightarrow x = \frac{2T}{T+1} \tag{15}$$

Then, the total number of samples needed to generate p_n using Equation 11 is determined by the number of times a portion of p_n needs to be added to the original starting point to include all the features of p_n:

$$UBAC(T) = \frac{1}{1-x} - 1 \tag{16}$$

$$= \frac{1}{1-\frac{2T}{1+T}} - 1 \tag{17}$$

$$= \frac{1+T}{1-T} - 1 \tag{18}$$

This equation represents an *upper bound* for the attacker cost (*UBAC*) since we assumed that the two points have completely disjoint feature sets. Using the equation, for a threshold of $T = \frac{1}{2}$, an attacker must generate 2 data mines to merge target points p_0 and p_3. For $T = \frac{9}{10}$, 18 data mines must be created.

Removing Assumption: "Similar Sizes"

During the upper bound analysis, we made the assumption that the two target points, p_0 and p_n, had roughly the same number of features. However, if the points had drastically different numbers of features, the adversary would have to first generate an extra series of points to scale the smaller point's feature set to a similar size as the larger point. Then, she may proceed using the same approach outlined above. The process of scaling up the smaller point requires creating points in the following manner:

$$p_i = p_{i-1} + x_i p_n \tag{19}$$

Where x_i represents a portion of the features in p_n. In this case, x_i must be selected based on T and the relative sizes of p_n and p_{i-1}:

$$J(p_{i-1}, p_i) = \frac{p_{i-1} \cap (p_{i-1} + x_i p_n)}{p_{i-1} \cup (p_{i-1} + x_i p_n)} \tag{20}$$

$$= \frac{|p_{i-1}|}{|p_{i-1}| + |x_i p_n|} = T \tag{21}$$

$$\Rightarrow x_i = \frac{(1-T)|p_{i-1}|}{T|p_n|} \tag{22}$$

Unlike Equation 15, x_i is dependent on the size of p_{i-1}. This means that as we scale the original (and smaller) point, p_0, we can add larger and larger portions of p_n to the next mine. In fact, the total number of mines to scale p_0 to the same size as p_n is logarithmic in the size of $\frac{p_n}{p_0}$. The base of the logarithm is $\frac{1}{T}$ as the portion of p_n that can be used for p_i is inversely dependent on the similarity threshold.

Removing Assumption: "$MinPts = 2$"

During the analysis, we made the assumption that $MinPts = 2$. This effectively made every point a core point which simplified the algorithm to generate data mines. In fact, the original algorithm works for $MinPts$ values 2 and 3 (see Figure 27a) since the point, p_i, and its predecessor and successor, p_{i-1} and p_{i+1}, are in the T-neighborhood of p_i.

Figure 27b depicts a geometry that the attacker can use to make each p_i a core point when $MinPts$ is 5. For odd values of $MinPts$, the attacker can generate similar geometries with $k = \frac{MinPts-1}{2}$ equally spaced points on both sides of p_i. The distance between these points should be such that:

$$\forall i \in [k,n) : J(p_{i-k}, p_i) = T \tag{23}$$

For the Jaccard distance, this means that:

$$\forall i \in [0,n) : J(p_i, p_{i+1}) = \sqrt[\frac{MinPts-1}{2}]{T} \tag{24}$$

Then, the geometry of p_i and its neighbors, p_{i-1} and p_{i+1}, is the same as Figure 27a except that T has been replaced with $T' = \sqrt[\frac{MinPts-1}{2}]{T}$. Therefore, we can substitute T' into Equation 18 to determine the number of mines required to merge two clusters as a parameter of both T and $MinPts$:

$$UBAC(T, MinPts) = \frac{1+T'}{1-T'} - 1 \tag{25}$$

$$= \frac{1+\sqrt[\frac{MinPts-1}{2}]{T}}{1-\sqrt[\frac{MinPts-1}{2}]{T}} - 1 \tag{26}$$

There is a caveat to the above approach: it requires that both p_0 and p_n be core points in their respective clustering. This can be accounted for by adding enough data mines to the T-neighbors of p_0 and p_n such that they are now core points. This requires at most $2 * MinPts$ more data mines be added. This additive constant is left out for simplicity.

Remediation

Before applying DBSCAN to her dataset, the defender has a chance to perform sanity checks on the dataset to determine if any apps should be removed. We explore two ways that the attacker could select apps for removal: 1) remove apps at random with some probability or 2) train a classifier to identify apps to remove.

Random Remediation

In the previous section, we describe an optimal mechanism to merge clusters using the fewest data mines possible. However, the bridges created by this approach are brittle – there is no redundancy

in the chain so the omission of a single data mine nullifies the attack. A defender may leverage this fact to make her clusters more robust against tampering. Specifically, she omit each app with some probability, p. Then, the probability of unlinking a chain of n data mines is:

$$Pr[\text{Preserving chain}] = (1-p)^n \tag{27}$$
$$\Rightarrow Pr[\text{Unlinking chain}] = 1-(1-p)^n \tag{28}$$

Outlier-based Remediation

Outlier detection is commonly used when analyzing data to identify points that are characteristically different from others in the dataset. For example, we can identify observations that are statistically unlikely given the population's mean and standard deviation. There are many different outlier measurements, some are based on neighborhood relations, others are based on local densities [**?**], and others are based on angles between points [**?**]. Since data mines are constructed to minimally span the gaps between clusters, we hypothesize that outlier measurements can identify these points. Rather than rely on a single outlier measurement to predict whether an app is a data mine or not, we instead propose a supervised approach using an ensemble of outlier measures. Specifically, we propose computing outlier measurements on a data set that we have tampered with, training a classifier to identify the data mines we injected, and then applying the classifier to data sets that may have been tampered with by an adversary. For our initial experiments, we compute the following outlier measurements for each point to use as features for our classifier:

- The number of neighbors in the $T^{\frac{1}{4}}$, $T^{\frac{1}{2}}$, T, T^2, T^3 neighborhoods, where T is the DBSCAN clustering threshold.
- The angle between the two nearest-neighbors.
- The variance in the angle between all pairs of points in the same cluster (similar to the Angle-base outlier factor [**?**]).

In general, outlier measurements can be fairly computationally expensive, so we select measurements that are tractable for our target data set. However, we can easily add other outlier measurements to our ensemble with only the additional computational cost.

8.6 Dataset

The full dataset used for AnDarwin [**?**] consists of 265,359 apps crawled from 17 Android markets. From those apps, AnDarwin extracted a total of 90,144,000 semantic vectors which it then clustered into 2,952,245 features. AnDarwin clustered the apps into 28,495 clusters of which 4,679

clusters contain apps from more than one owner. Ownership is determined using the methodology described in Section 8.5.

We develop our attacks using two subsets of the full dataset. The first (DS0) is 273 clusters selected at random from the 28,495 clusters identified by AnDarwin. The second (DS1) is the 4,679 clusters that AnDarwin identified as having apps from more than one owner. A breakdown of the two datasets is shown in Table 13. We note that we were unable to precisely replicate the original results of AnDarwin: we identified 3,705 multi-owner clusters in DS1. We hypothesize that this is due to our list of developer accounts and signing keys being more complete (ours included several more months of crawling).

	DS0	**DS1**
# Apps	1,394	29,788
# Clusters	273	4,679
# Single-Owner Clusters	229	974
# Multi-Owner Clusters	44	3,705
# Plagiarizing Apps	196	12,452

Table 13: Statistics for the two experimental datasets based on subsets of the AnDarwin clusters.

8.7 Evaluation

In this section, we evaluate the effectiveness of the confidence attack. Specifically, we generate a series of cluster merges using each of the ordering algorithms (Section 8.5) and then merge clusters by generating data mines (Section 8.5). For the sake of thoroughness, we evaluate the effectiveness of these attacks to "completion," when the attacker has merged all clusters into a single cluster. In reality, an attacker would likely not perform such an attack and would instead have a budget on the number of data mines or a specific goal (e.g. merge clusters X, Y, and Z).

First, we look at the number of data mines required to perform the attack to "completion." Then, we evaluate how the clustering degrades using our clustering performance metrics. Next, we perform an analysis to detect inadvertent merges. Then, we evaluate a potential defense against this attack: increasing T and $MinPts$, and evaluate the cost to both the defender and the attacker. Finally, we look at two ways to remediate the clusters and explore how well they recover the original plagiarism detection accuracy.

Data Mines

The number of data mines required to merge all the clusters into a single cluster for each of the merge ordering algorithms is shown in Table 14. This required a total of 272 and 4,678 merges for DS0 and DS1, respectively. Interestingly, the number of data mines is quite similar across all algorithms except for Decreasing Cluster Similarity which required approximately 35% more data mines than the other algorithms. We hypothesize that this is because most clusters are pairwise

dissimilar but some clusters may have apps that share a few features. This non-zero similarity will cause these cluster pairs to be merged before cluster pairs with no overlap, even if the apps in each cluster are drastically different sizes. This leads to more scaling data mines overall.

Algorithm	**DS0**	**DS1**
Cluster Sim., Decreasing	1,145	20,363
Cluster Sim., Increasing	1,100	14,454
Random	909	16,010
Cluster Size, Decreasing	869	15,574
Original Size, Decreasing	869	14,780
Author Size, Decreasing	866	15,781
Cluster Size, Increasing	840	15,574
G.P. Accuracy	830	N/A
Nearest-neighbor	818	15,085

Table 14: The number of data mines to merge all clusters into a single cluster ("completion").

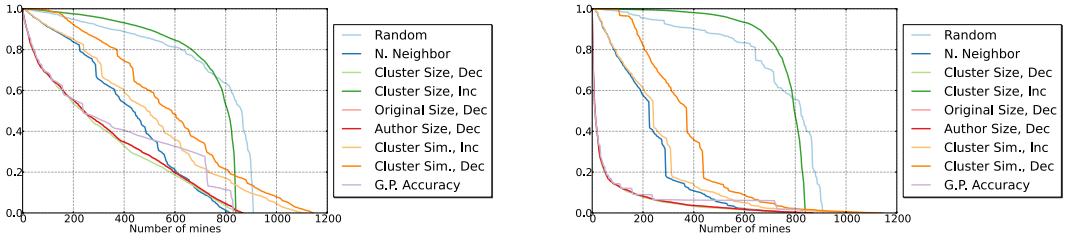
Interestingly, the number of data mines required to merge all clusters is more than half the number of apps in the original dataset. This means that the attacker has to inject a significant number of mines relative to the size of the original dataset in order to merge all clusters to completion. However, the number of data mines required is not a function of the number of apps in the dataset; it is a function of the number of clusters. Using Equation 18 and the DBSCAN parameters of $T = 0.5$ and $MinPts = 2$, the number of mines to required to merge two arbitrary clusters is 2. Therefore, to merge the 273 clusters (DS0), a total of $2 * 272$ data mines should be needed. The discrepancy between the theoretical and actual values is due to the fact that not all apps have the same number of features. As a result, additional mines are required (as described in Section 8.5) to match up the apps' feature set sizes, increasing the total number of mines to merge two clusters.

Clustering Degradation

In Figure 28, we plot the values of the clustering performances metrics described in Section 8.5 for DS0. Figure 29 shows the same plots for DS1. The metrics compare the clustering after some number of data mines have been added to the original clustering. Each metric has 1.0 as a perfect score. We can make a number of interesting observations about these plots:

First, the relative ordering of merge algorithms is largely consistent across the generic clustering comparison metrics. Decreasing Cluster Size and Decreasing Original Size tend to do the best while Increasing Cluster Size and Random do the worst, from the attacker's perspective. These metrics are all roughly based on the number of points that are clustered correctly, a property which degrades the most quickly if the larger clusters are merged first.

Second, the results for the application-specific metric, the plagiarism detection accuracy, are mostly consistent with the merge algorithm effectiveness as determined by the generic clustering comparison metrics. Noticeably, the Greedy Pessimal algorithm that merges the adversary's best



(a) Homogeneity

(b) Adjusted Rand Index

(c) Adjusted Mutual Info

(d) Plagiarism Detection Accuracy

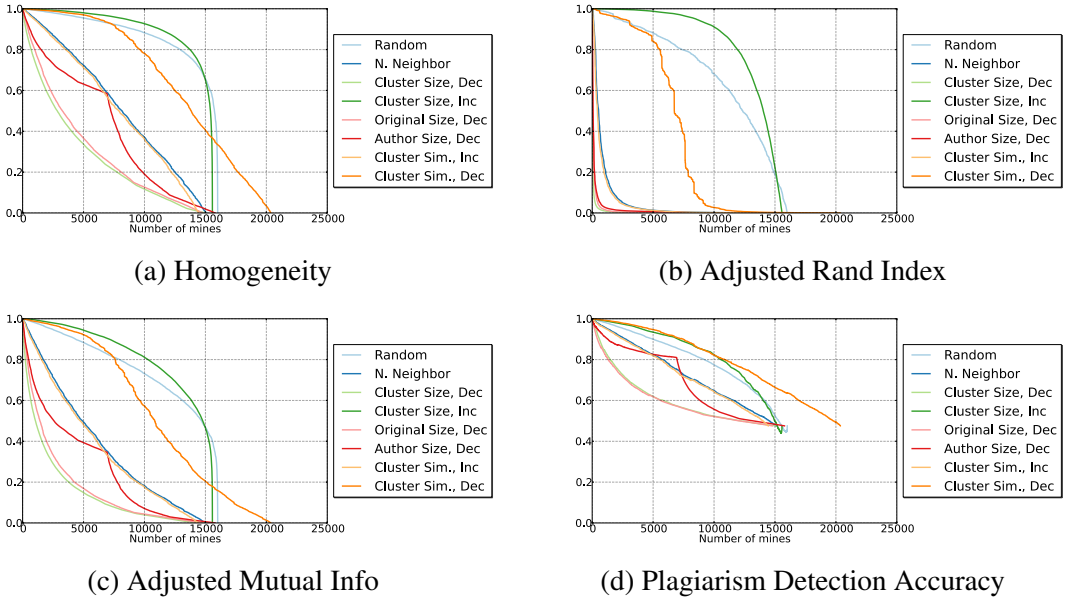
Figure 28: Cluster degradation plots for `DS0`. These show how the four clustering performance metrics degrade as a function of the number of data mines the attacker has injected into the dataset. From an attacker's perspective, algorithms with less area under the curve are better since they drop the clustering performance quicker.

pair at each stage of merging is only just more effective than the Decreasing Cluster and Author Size algorithms. Given it's computational complexity, we do not compute the Greedy Pessimal merge order for `DS1` but we expect it to be similar to the Decreasing Cluster Size algorithm.

Third, increasing similarity and nearest neighbors have near-identical performance. This is due to a majority of the apps being mutually dissimilar. If we select a cluster at random as the seed cluster and merge in order of it's nearest neighbors, we may find a few similar clusters but the majority will be dissimilar. Therefore, the majority of pairs in both increasing similarity and nearest neighbors will be dissimilar. Since we order the clusters for processing in the same way each time, we produce near identical cluster merging orders.

Fourth, the Adjusted Rand Index metric seems to be the most sensitive to the cluster poisoning. After only a few hundred mines for the big to small merge ordering, the metric drops to close to zero while the other metrics remain above 0.8. This presents an opportunity for the defender to quickly detect cluster poisoning. If the defender were performing incremental clustering using DBSCAN over a period of time, she could compare today's clustering to previous clusterings to see how much they differ. If the adversary acts too quickly, a large drop in the Adjusted Rand Index could alert the defender.

Finally, the plagiarism detection accuracy for `DS1` only degrades to around 50% accuracy, despite all the clusters being merged. When we manually investigated the "original" apps in the final cluster that were from the same author (Section 8.5), we found several prolific plagiarizers. Specifically, we found seven developers with more than 100 apps each (the most was 798 apps). These



(a) Homogeneity

(b) Adjusted Rand Index

(c) Adjusted Mutual Info

(d) Plagiarism Detection Accuracy

Figure 29: Cluster degradation plots for `DS1`. These show how the four clustering performance metrics degrade as a function of the number of data mines the attacker has injected into the dataset. From an attacker's perspective, algorithms with less area under the curve are better since they drop the clustering performance quicker.

are likely plagiarizers who downloaded as many apps as possible and uploaded the apps to their own accounts. These prolific plagiarizers, which led to a prolific merged author, also explain why the plots for the Author size merge algorithm have two phases: the many apps from the prolific author are spread across many clusters initially.

Inadvertent Merges

In our implementation of the attacks, we split the stages of merge ordering and data mine generation into two separate phases. We compute the pairwise similarity matrix for the points and the clustering before running the merge ordering algorithms but we do not update the matrix or rerun the clustering after each data mine is added. This could lead to *inadvertent merges* where, when merging two clusters, some of our data mines cause a third or more cluster to be merged in with the two that we intended to merge and could cause us to "overspend" as an attacker.

Rather than merge the two stages of the attack, we instead post-process a pairwise similarity matrix that contains all the original points and the data mines. First, we record the number of clusters found when clustering the original points. Then, we add a set of data mines forming a bridge to the matrix, recompute the clustering, and record the number of clusters. We repeat this process for each of the bridges. Each bridge should merge exactly two clusters so the number of clusters should decrease by one for each bridge added. If the number of clusters decreases by more than one, an inadvertent merge has occurred.

We ran the methodology described above on all the merge algorithms for DS0 and found no inadvertent merges. For DS1, there was one inadvertent merge for the Increasing Cluster Size ordering algorithm. Figure 30 compares the expected number of clusters vs the observed number of clusters for the ordering algorithm. Here, we can see there was one inadvertent merge around the $1,400^{th}$ merge and that it was corrected after about 3,000 merges when the algorithm tried to merge a cluster that had already been inadvertently merged.

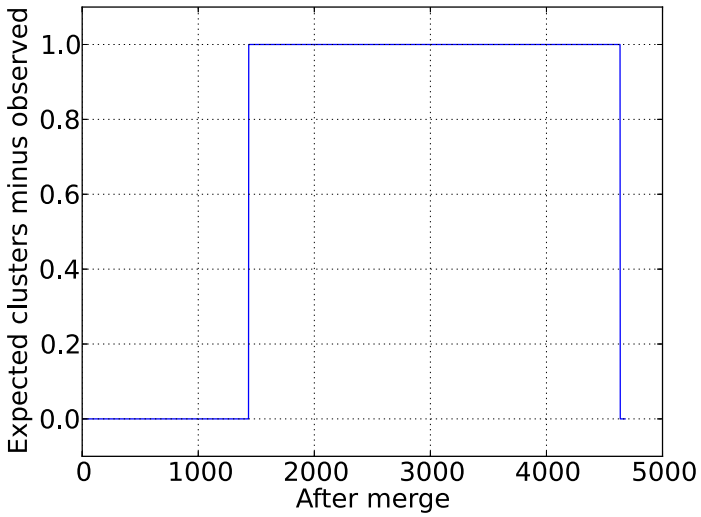


Figure 30: Inadvertent merge from the Increasing Cluster Size ordering algorithm for DS1. The merge happened after around the $1,400^{th}$ merge and was corrected when the inadvertently merge cluster was supposed to be merged with a cluster that it was already merged with.

The chosen feature space for Android apps is very sparse so inadvertent merges are unlikely to occur. For DBSCAN-based tools in a more dense feature space, inadvertent merges may help reduce an attacker's cost.

Attacker and Defender Costs

In this section, we explore one possible defense against the cluster merging attack: increasing T and $MinPts$. By increasing T and $MinPts$, the defender can increase the number of data mines the attacker must generate in order to merge two clusters (Equation 26). This increases the cost to the attacker as she will have to generate many more data mines to bridge clusters. However, this is not without cost to the defender. If the defender increases $MinPts$ from 2, she will no longer be able to detect apps that have been copied just once. Drastically increasing $MinPts$ will allow her to only detect frequently copied apps. If the defender increases T, she may miss some copies. A defender must balance her own cost against that of the attacker if she is to use this as her defense.

In Table 15, we show the attacker's cost (Equation 26) computed for different values of T and $MinPts$. From this table, we can see that the attacker cost does not increase when $MinPts$ is increased from 2 to 3. This is because the chaining geometry always creates mines such that

the number of points in the T-neighborhood of a point is 3. At higher values of T and *MinPts*, the attacker cost is relatively high; she must generate more than 50 points in order to merge two clusters.

	MinPts				
T	2	3	5	11	19
0.5	2.0	2.0	4.828	13.45	24.98
0.6	3.0	3.0	6.873	18.59	34.25
0.7	4.667	4.667	10.24	27.05	49.47
0.8	8.0	8.0	16.94	43.82	79.67
0.9	18.0	18.0	36.97	93.92	169.8

Table 15: The attacker's cost for various values of T and *MinPts*, as measured by the number of points required to merge two clusters (Equation 26).

In Table 16, we show the defender's cost computed for different values of T and *MinPts* for `DS0`. A goal of AnDarwin is to find plagiarized Android apps. Therefore, we measure the defender cost as the number of apps that are no longer detected as plagiarizing with the new parameter values. For this reason, the cost is zero when $T = 0.5$ and $MinPts = 2$. Originally, we classify 196 of the 1,394 apps in `DS0` as plagiarizing. With only minor increases in T and *MinPts*, we can see that the number of plagiarizing apps drops by about 10% (see Section 8.8 for a discussion of whether all these apps are indeed plagiarizing in the first place). Table 17 shows the defender's cost for `DS1` where we originally classify 12,452 of the 29,788 app as plagiarizing.

	MinPts				
T	2	3	5	11	19
0.5	N/A	14	28	62	70
0.6	12	25	42	75	81
0.7	39	53	69	92	93
0.8	52	63	75	92	93
0.9	137	146	159	174	174

Table 16: The defender's cost for various values of T and *MinPts*, as measured by the number of plagiarizing apps no longer detected as plagiarizing for `DS0`.

	MinPts				
T	2	3	5	11	19
0.5	N/A	1,126	2,939	5,652	7,521
0.6	1,874	3,045	4,792	7,182	8,817
0.7	3,747	4,927	6,554	8,612	9,862
0.8	5,670	6,863	8,257	9,738	10,745
0.9	7,538	8,660	9,713	10,724	11,265

Table 17: The defender's cost for various values of T and *MinPts*, as measured by the number of plagiarizing apps no longer detected as plagiarizing for `DS1`.

Finally, in Table 18, we compare the attacker's and the defender's costs for DS0. Specifically, we compute the defender's cost divided by the attacker's cost:

$$Rel_{cost}(T, MinPts) = \frac{\text{Missed Plagiarism Detections}}{UBAC(T, MinPts)} \tag{29}$$

When selecting T and $MinPts$, the defender wants to minimize this value while balancing her own cost. If she selects $T = 0.9$ and $MinPts = 19$, she can minimize Rel_{cost} but she will only be able to detect plagiarizing apps for apps that have at least 19 copies and that are all 90% similar. That is, consulting Table 16, she will fail to find 174 plagiarizing apps.

	MinPts				
T	2	3	5	11	19
0.5	N/A	7.0	5.799	4.61	2.802
0.6	4.0	8.333	6.111	4.034	2.365
0.7	8.357	11.36	6.735	3.401	1.88
0.8	6.5	7.875	4.426	2.099	1.167
0.9	7.611	8.111	4.3	1.853	1.024

Table 18: $Rel_{cost}(T, MinPts)$ at various levels of T and $MinPts$ for DS0.

	MinPts				
T	2	3	5	11	19
0.5	N/A	563.0	608.687	420.222	301.065
0.6	624.667	1,015.0	697.223	386.271	257.457
0.7	802.929	1,055.79	639.764	318.39	199.342
0.8	708.75	857.875	487.303	222.219	134.869
0.9	418.778	481.111	262.7	114.187	66.3256

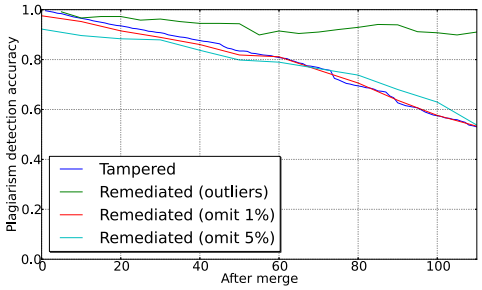
Table 19: $Rel_{cost}(T, MinPts)$ at various levels of T and $MinPts$ for DS1.

Based on this analysis, we find that increasing T and $MinPts$ is an insufficient defense for preventing a confidence attacks that seeks to poison the clustering.

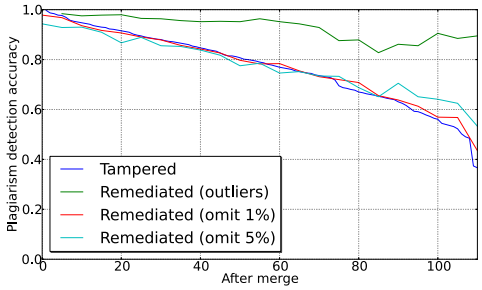
Remediation

In order to test our proposed clustering remediations, we partition DS0 into two partitions: A) 700 apps forming 153 clusters and B) 694 apps forming 120 clusters. The partitioning was performed randomly by cluster until the number of apps in each partition was approximately equal. In this section, we explore how the proposed remediation methods change the plagiarism detection accuracy after some number of merges. We generate mines using two different merge orderings: random and decreasing original size. We select these two algorithms as they are representative of a weak and strong adversary (when the adversary's goal is to degrade plagiarism detection accuracy).

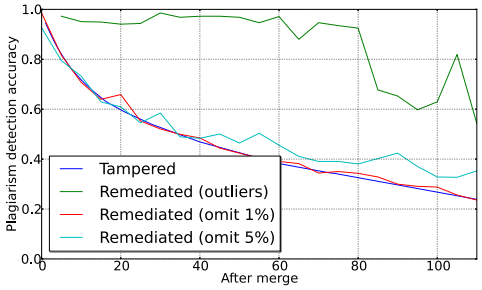
First, we tested the random remediation methodology using different levels of p, the probability of excluding a given point, and varying the number of merges in the tampered data set. We tested p values between 5% and 30%, inclusively, in 5% increments and specifically at 1%. We varied the number of merges between 0 and 110, inclusively, in increments of 5 merges. We then compute the plagiarism detection accuracy at each of these 161 values for both partitions of the dataset and both merge algorithms. In Figure 31, we plot two columns from this table: the accuracy at 1% and 5%, for each of the partitions of the dataset.



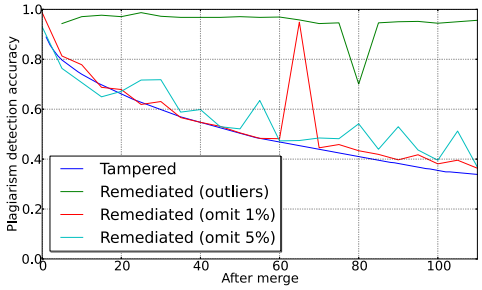
(a) Partition A, Random.



(b) Partition B, Random.



(c) Partition A, Decreasing Original Size.



(d) Partition B, Decreasing Original Size.

Figure 31: The plagiarism detection accuracy with and without remediation on two partitions of DS0. The top two plots show plagiarism detection accuracy when the adversary merges clusters randomly, and the bottom show when she merges based on the number of original apps in the cluster. The tampered curves show how accuracy degrades without remediation. For the outlier remediation curve, the presumed number of merges in the training partition matched the actual number of merges in the testing partition.

As we can see from the Figures, remediation by randomly removing points with some probability does not improve the plagiarism detection accuracy in most cases. Surprisingly, for partition B, random remediation against the Decreasing Original Size merge algorithm does almost as well as the outlier-based approach. This was a statistical anomaly, when we rerun the experiment 100 times with the same parameters, we only observed a similar plagiarism detection accuracy twice. Interestingly, once a large number of clusters have been merged, the random remediation methodology may be marginally useful.

Next, we test our proposed outlier-based remediation approach. First, we compute outlier measurements for apps in the training partition, varying the number of clusters that were merged before computing the features. We then train a classifier with the presumed number of merges

and test the classifier on the testing partition, varying the number of actual merges. We varied the presumed number of merges between 5 and 110 and the actual number of merges between 0 and 110. In both cases, the ranges were inclusive and we computed the results in increments of 5 merges. In Figure 31, we plot the diagonal of the matrix which is the ideal case when the presumed level of tampering matches the actual level of tampering, for each of the partitions of the dataset (and using the other partition for training).

From the Figures, we can see that the outlier-based remediation approach is fairly successful at recovering the untampered plagiarism detection accuracy under the ideal circumstance where the presumed amount of tampering the actual amount of tampering. In fact, as long as the presumed amount of tampering is less than the actual amount of tampering, the outlier-based remediation approach does well (see Figure 32). Surprisingly, training a classifier on data with just five merges leads to near-perfect remediation, regardless of the number of merges present in the actual dataset.

Another interesting experiment to pursue in future work is to train a classifier assuming one attacker merge algorithm and test on another. This is a more realistic scenario as we are unlikely to know the attacker's strategy ahead of time.

8.8 Discussion

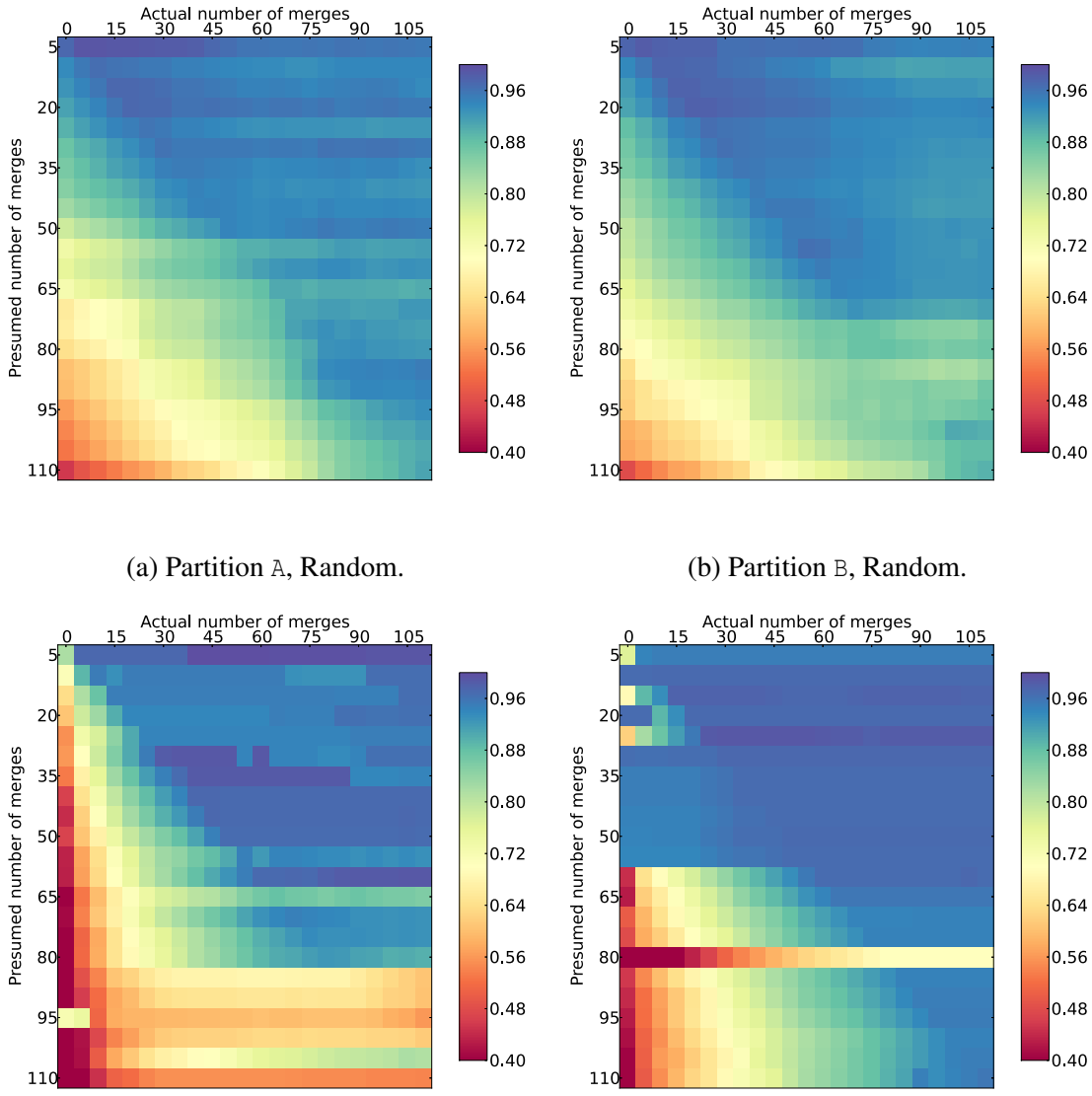
Attack Feasibility

In our threat model, we assumed that the attacker has perfect knowledge: she knows the complete dataset, the feature space, the algorithm, and the algorithm's parameters. We explore this scenario as the worst case behavior both for the sake of general insight into the problem and for scoping the specific vulnerability of clustering tools based on DBSCAN. Further, in the worst case, the attacker is an insider who, as an insider, does have perfect knowledge.

In the specific case of our AnDarwin application, the dataset is comprised of publicly available applications crawled from Android Markets; this could be replicated by an attacker. Even if such an attacker's collection did not perfectly match the defender's collection, our mechanism for generating bridges between clusters still applies. Admittedly, the attack would likely be suboptimal; characterizing the degree of suboptimality as a function of matching the defender's data collection is an interesting problem for future work.

Merge Algorithms

In Section 8.7, we evaluated the clustering performance degradation using one instance of each of the merge algorithms described in Section 8.5. However, for each merge algorithm, there are many instances of the attack. Some of these instances will outperform the others in terms of how quickly they degrade the clustering performance. Ultimately, there are "optimal" attacks that degrade the clustering performance the fastest for a particular metric with the fewest points. Naively,



(a) Partition A, Random.

(b) Partition B, Random.

(c) Partition A, Decreasing Original Size.

(d) Partition B, Decreasing Original Size.

Figure 32: Plagiarism detection accuracy for the various levels of presumed and actual tampering for the two partition of DS0 for two different merge algorithms.

discovering optimal attacks is a combinatorial problem as every ordering of pairs of clusters must be considered. There may be greedy algorithms that approximate the optimal ordering.

Suboptimal Data Mines

In Section 8.7, we evaluated how well our outlier-based remediation approach was able to remove data mines from the dataset to recover the original plagiarism detection accuracy. We built and

tested our classifiers for data mine detection using outlier features that are computed for clusters that have been optimally merged using the fewest number of data mines possible. However, an adversary may not attempt to minimize the number of data mines she uses. In fact, based on the results of our remediation experiments, the adversary should not use optimally-placed data mines if she wishes to avoid detection. An interesting problem for future work is to explore the degree of suboptimality required to evade the outlier-based remediation approach. Two potential approaches include simply generating data mines paths with higher values of T and $MinPts$, and adding jitter to "widen" the data mine paths.

Plagiarizing apps

For evaluating the defender cost of altering the DBSCAN parameters T and $MinPts$ in Section 8.7, we assumed that the original clustering was correct. Specifically, we assumed that all the apps that are identified as plagiarizing with the original parameters are indeed plagiarizing. This, however, is not necessarily the case. False alarms in the original clustering will increase the defender's cost even though they are false alarms. In fact, we could be improving the clustering with the different values of T and $MinPts$. Knowing the ground truth clustering of this dataset is outside the scope of this work and the evaluation was designed to measure, in the worst case, the cost to the defender when T and $MinPts$ are set to increase the attackers cost.

8.9 Future Work

In this section, we specifically investigated AnDarwin for the methodology for generating mines and in the evaluation. However, the concepts of bridging gaps between clusters applies to all clustering tools that are based on the original DBSCAN algorithm. The exact construction of data mines varies with the distance metric and the feature space. Some examples of where the confidence attack could be used against a DBSCAN-based tool include subverting network traffic classification [**?**] and query clustering for search engines [**?**]. In future work, we plan to investigate how the number of data mines to bridge clusters varies with other feature spaces and distance metrics.

Assuming that adversaries will tamper with datasets using the methodology described in the paper, the next question is whether this tampering can be prevented or, at least, detected. In Section 8.7, we evaluated whether the defender could simply tweak the DBSCAN parameters to create a suitable defense and found that it was insufficient. One possibility would be to change the DBSCAN algorithm to use an alternative to single-linkage when merging clusters. A new version of DBSCAN based on complete-linkage would be immune to bridging but may miss some clusters.

We earlier described an alternative way to attack a clustering algorithm, an *evasion attack*. In this attack, an attacker seeks to suppress the creation of clusters involving her data. Most likely, this would be achieved by obfuscating the data in feature space. More generally, an attacker may wish to break up existing clusters by "deleting" points that exist in the dataset. In future work, we

plan to explore how an attacker can strategically delete or avoid the creation of certain data points in order to prevent clusters from forming.

Finally, there have been a number of improvements suggested to the original DBSCAN algorithm. In particular, C-DBSCAN [**?**], adds constraints to the clustering algorithm. We plan to explore whether C-DBSCAN and other DBSCAN-based algorithms remain vulnerable to the concerns raised here.

9 Conclusions and Future Work

9.1 A Summary of Results to Date

In conclusion, we have:

- Exhaustively investigated the impact of adversarial label tampering on supervised machine learning, and in the process generated a durable body of re-usable Python code for future such investigations (Section A).
- We have demonstrated the counter-intuitive and alarming result that there exist label tampering attacks which are very effective (in the sense of decreasing test set accuracy near linearly with the number of points attacked) while being nearly undetectable by the usual training data cross-validation tests (Figure 6 and Section 4.3).
- The effectiveness of the various attacks we invented are, of course, worrisome. So we also invented, described (Section 5) and thoroughly quantitatively investigated (Section 6) "Ensembles of Outlier Measures", a method for building a meta-model for detecting tampered data. Further, we have shown how to use EOM to remediate tampered data by detecting, and correcting, the labels of suspect data points.
- In addition to working up a mechanism for individually detecting tampered points, we also created a method (Section 7) for "quantified paranoia", that is, a statistically principled mechanism for deciding whether a dataset as a whole has been tampered with, regardless of whether we can detect any individual bits of tampering with confidence. In tests we demonstrated (Table 12) that our quantified paranoia test does indeed find evidence of very limited tampering, even the tampering invisible to cross-validation, while not reacting to statistically similar data which is untampered though indeed different.
- Finally, we have shown that the general principles behind Ensembles of Outlier Measures as a remediation technique are surprisingly and satisfyingly general. That is, in Section 8 we develop an entirely different sort of adversarial tampering problem, one in which an adversary attempts to undermine an *unsupervised* machine learning method called DBSCAN by adding spurious elements to a data set. So the nature of the data, the nature of the analytic, and the nature of the attack are all very different from the supervised machine learning work investigated in most of CADA: yet an ensemble of outlier measures nonetheless turns out to an effective means of detecting and removing the attack points, returning the clustering algorithm to its unattacked accuracy (Figures 31 and 32).

9.2 Next Steps

CADA was a brief project, just 1.5 years, and remarkably productive in terms of generating code, experiments, insights, and new ideas. As a result, we generated a long list of related ideas that we

were able to investigate only shallowly, or not at all. Here is a partial list, as fuel for the various projects to follow CADA:

- In Section 4.3 we describe a number of metrics for describing the overall merits of various attack algorithms, but as noted there, we started but did not finish a search for a metric that more strongly weights results when the budget is low, the idea being to reward attacks that are quickly effective without having to separately investigate and report the effectiveness of an attack at 5% tampering, 10% tampering, etc.
- In Section 4.5 we made the unexpected observation that Nearest Neighbor Cohort, a method *designed* to be better at evasion attacks than plain Nearest Neighbor, in fact is not. As observed there, "Chaos is somehow superior to consistency here, which bears further investigation". Further, as noted in Section 8.9, investigation of evasion attacks seems a promising next step for the clustering work as well.
- A point we noted repeatedly, though anecdotally, when developing the idea of using outlier features to detect tampered data (Section 6) is that outlier features are more robust to both labeling tampering and feature drift than the sort of primary features that fueled the ideology or production inspection datasets. This is important, as otherwise an adversary could launch a meta-attack on our meta-defense and try to tamper with the tamper detection model. We have the beginnings of an explanation for why this is difficult to do (rooted in the fact that outlier features depend on critically on one's *neighbors* in a way that the base features do not), but it would be useful to develop the theory and conduct some quantitative experiments.
- In the final discussion (Section 6.4) of the use of Ensembles of Outlier Measures (EOM) for detecting and remediating tampered data points, we noted that the sensible next step is to use the existing experimental harness to dig into very detailed questions of how defense assumptions play against actual attacks. In particular, we'd like to develop metrics and post-processing tools to illuminate:
  - Worst case analysis: what 4-tuple combination of "presumed attack, presumed budget; actual attack, actual budget" was worst for the defender? What combination was worst for the attacker? What trends are observable?
  - Similarly, what presumed attack was most robust for defender, over the range of known possible attacks? Which actual attack was most robust for attacker, over the range of possible defenses?
  - Which outlier features were most useful over all?
  - Which outlier features were most useful when unremediated accuracy was high? Was low? Is there a difference?
  - If we look at the relative efficacy of the outlier features for the 4-tuple where the defender does worst, or where the defender does best, what insight might that give us as to how to improve or add to our current set of outlier features?
- In Section 7.6 we presented results indicating that a test based on PBF was able to find very subtle label tampering. It is at least possible, however, that the same test would respond

positively to normal feature drift, and is more a "change in data" test than a "label tampering" test.

To investigate this, we started but were unable to finish analysis an analysis of s500 that would have mirrored that of ideology. The difference is that we could have additionally compared untampered from different years, and thus differentiated tampered data from drifting data. This would require work generalizing and scaling the PBF test to handle differences in data set size, feature scale, and other complications.

- In Section 7.6 we further asserted that Mallow's Distance provides a somewhat more sensitive measure of tampering than PBF, and provided an anecdotal heatmap illustration of this point. It would be attractive to flesh out Mallow's Distance, develop a theoretical grounding for where and how it would be expected to outperform PMF, and conduct experiments to explore that theory.

- In Section 8.9, in discussion of defending clustering against bridges, we note that the promising results we demonstrated are also potentially specific to the problem we studied, and thus it would be interesting to investigate how the number of data mines required to bridge clusters varies with other feature spaces and distance metrics.

- Or, similarly, how the number of data mines depends on the clustering scheme; a new version of DBSCAN based on complete-linkage would be immune to bridging but may miss some clusters and thus some plagiarized apps. Another angle would be to investigate various newly published DBSCAN variants, such as Constrained-DBSCAN[**?**].

- Finally, it is worth noting that all of the supervised machine learning in CADA was based on Ensembles of Decision Trees, EDTs. As elaborated upon in Section 2, this is because EDTs are generally very close to optimal, given the available features, and were already known to be robust to random label noise (Figure 10), which allow the effects of *adversarial* tampering to be more clearly studied.

  However, there were repeated hints throughout the project (and from another Sandia project titled "Mountain Creek"; see Philip Kegelmeyer, David Zage, or Curtis Johnson for details) that there may be a "robust yet fragile" property in play here. That is, there are indications that EDTs robustness to random tampering comes at the cost of *increased* vulnerability to adversarial tampering. Contrariwise, it might be the case that simpler machine learning models, such as linear classifiers, will start off much inferior to EDTs on complicated problems *but* degrade much more slowly as the problem is made harder by adversarial interference. It would be very interesting and useful to develop both theory and experiment to explore this point.

A Resources

This appendix documents some of the resources generated by this project that may be useful to future projects.

A.1 A Guide To The Source and Document Repositories

The CADA project was managed via a git repo hosted on Sandia's SRN, via the Sandia Github Enterprise installation, with an additional data store for big files. Various pointers and links (current as of November, 2014):

- Owners of the CADA git repo, with full admin rights: Philip Kegelmeyer and Tim Shead.
- Wiki: https://github.sandia.gov/cada/CADA/wiki. The CADA wiki is particularly well-populated, and is the place to start when digging into or searching concerning the CADA project. Links of particular interest are:
  - "Meeting Log", https://github.sandia.gov/cada/CADA/wiki/Meeting-Log, which contains detailed notes from each of the weekly CADA discussions, and
  - "Project Documents", https://github.sandia.gov/cada/CADA/wiki/Documents contains various project support documents. Some of them are various briefings or slides for internal discussions, but this link also hosts all of the project archival documents, from the initial proposal to the final quad charts for the LDRD office.
- Git repo for wiki: git@github.sandia.gov:cada/CADA.wiki.git
- Git repo for project: git@github.sandia.gov:cada/CADA.git. Note that the repo almost entirely consists of staff-specific "sandbox" directories (with the one exception of the documents/ directory where most, but not all, documents were produced). There is no "production" directory for central storage of mutually used code; code and data interfaces were instead managed informally, as needed.
- CADA Data file storage: brusand.ran.sandia.gov:/home/cada-data/. Tom Kroeger is the system administration for brusand.
- Mailing list: wg-cada@sandia.gov
- Metagroup: https://metagroup.sandia.gov/cgi-bin/metagroup.pl?group=wg-cada
- Managers of the Sandia github installation: Zach Benz and Cheston Bailon.

A.2 Python for Avatar Configuration and Control

The Avatar machine learning tools played a central role in our experiments: we used the Avatar file format for data interchange; we used Avatar to identify statistically-significant features for our clustering algorithms; and we used Avatar to train and evaluate hundreds of thousands of ensembles of machine learning classifiers for the experiments themselves. Thus, because our attack algorithms and experimental scaffolding were implemented in Python [**?**], it became important to provide good integration between Avatar and the rest of our Python code. To facilitate this, we created an *avatar* Python package with functionality for reading and writing Avatar format files, and running the Avatar tools. Highlights of the API include:

avatar.data The avatar.data module includes load() and dump() functions to read and write Avatar ".data" files. When loaded into memory, the files are stored as an *avatar.data.columns* object containing a collection of named columns, each backed by a NumPy [**?**] array. Converting the data into the widely used *numpy.ndarray* data structure made it possible to compute efficiently with the data and integrate with a large number of analysis toolkits such as Scikit-learn [**?**].

avatar.names The avatar.names modules includes load() and guess() functions for reading and generating Avatar ".names" files. The guess() function is particularly useful, as it duplicates the functionality of the Avatar *data_inspector* tool, with considerably better performance.

avatar.ensemble The avatar.ensmemble.build() function provides a convenient wrapper for running the *avatardt* program to train and optionally evaluate an ensemble of classifiers. The returned ensemble object provides methods to convert the avatardt results from logged-text into higher-level Python data structures. For example, accuracy metrics and confusion matrices are parsed from the avatardt stdout and returned as true scalars and arrays.

DISTRIBUTION:

1	MS 0359	D. Chavez, LDRD Office, 7911
1	MS 0899	Technical Library, 8944 (electronic copy)

Sandia National Laboratories